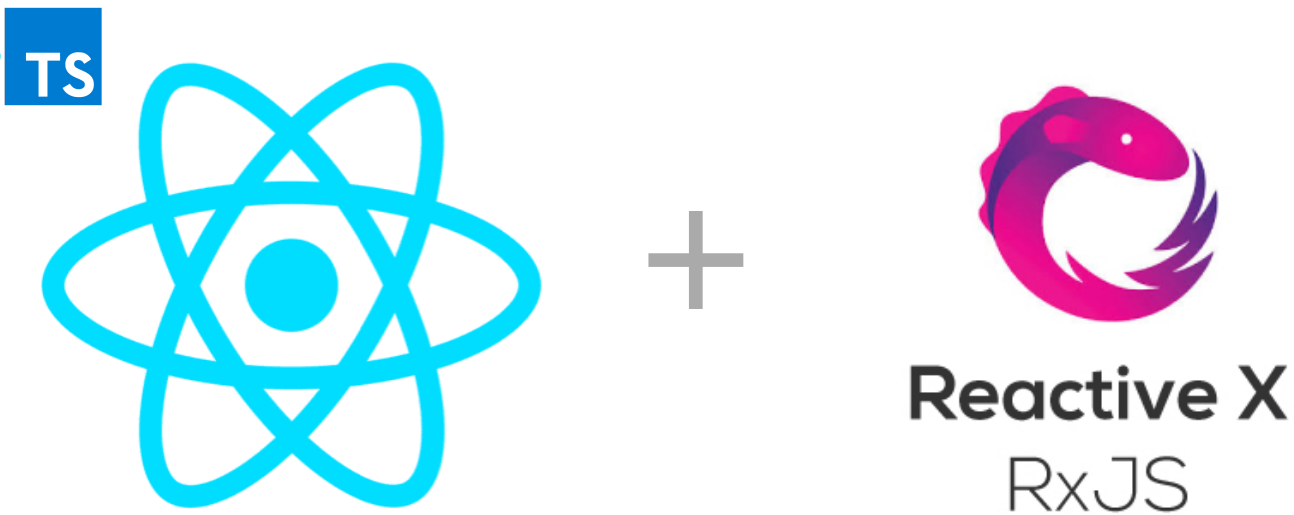
React Workshop

RxJS



Reactive JavaScript: **RxJS**

Understanding **observable** and **operator** concepts and how they play a role in application development is critical for both the Angular and **React** platforms.

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

Observables

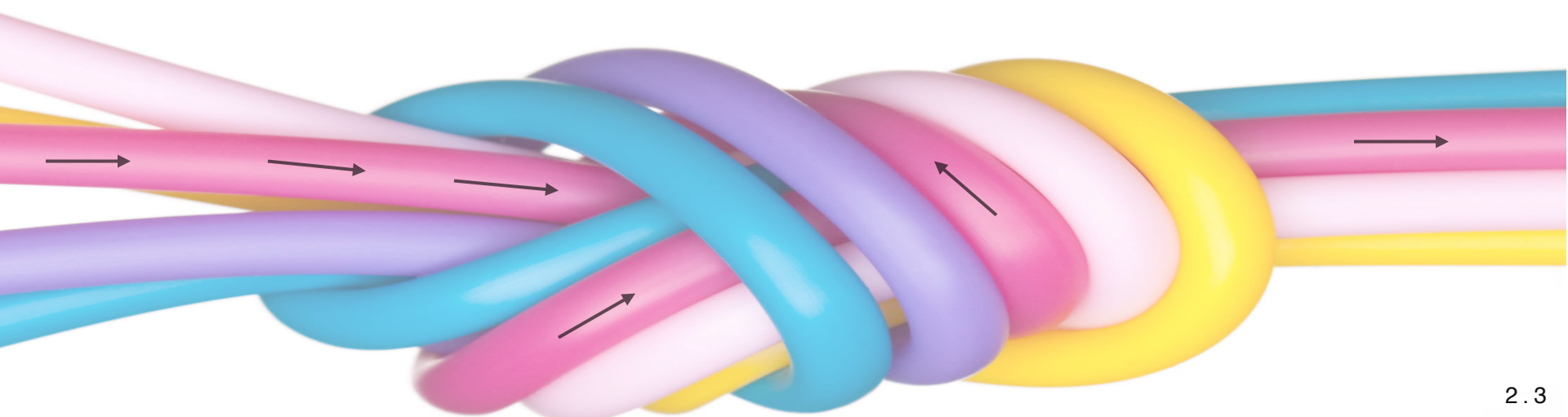
Observables allow us to change the way we build applications!

Instead of `**pulling**` data when needed, we will build applications that configure stream connections. With stream connections, data is `**pushed**` whenever it updates.

We now can build **REACTIVE** applications... where the UI reacts to data changes and stream emissions.

Observables

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

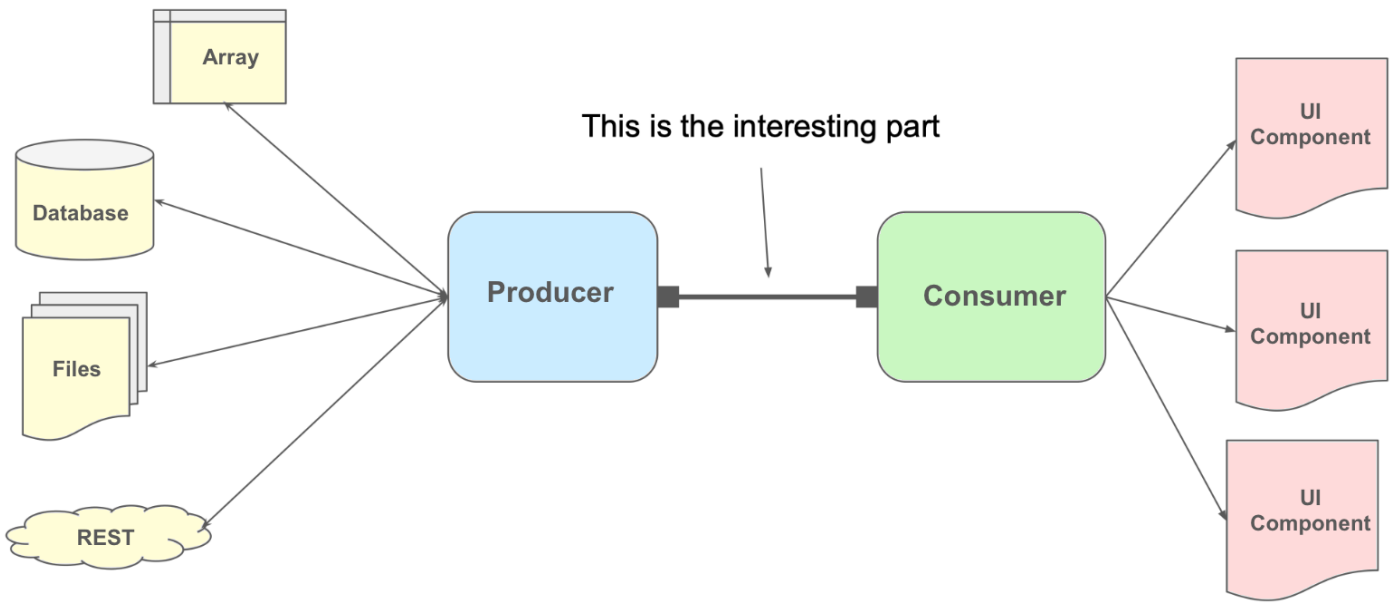


What we will cover?

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**?

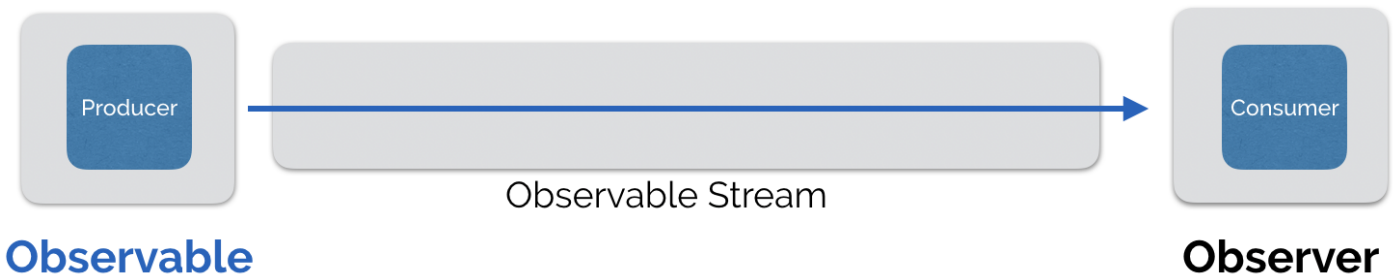
1

- Are **connections** 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**

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**; less 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**

Which one is a **Pure** Function ?

1

```
let lastName = '<your name>'

function makeFullName1(firstName): string {
  return `${firstName} + ${lastName}`;
}

function makeFullName2(firstName, lastName): string {
  return `${firstName} + ${lastName}`;
}
```

Push vs Pull

	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)

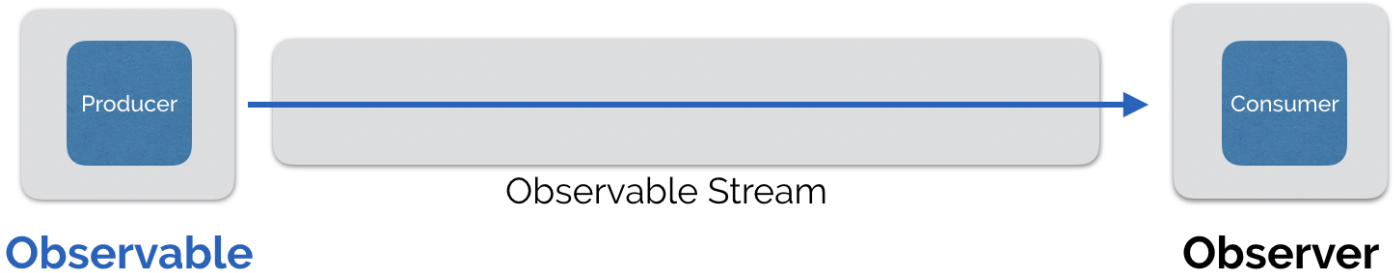
Speaker notes

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

Promise(s)

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

  // Producer Activity

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

});
```

Producer activity is started immediately

Both use 1st argument: Function

Observable(s)

```
const myStream$ = new Observable( (observer) => {

  // Producer activity
  // Notify external listeners using `observer`

  return () => {
        // Cancel producer activity, disconnect
  };

});
```

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

Observables are Functions

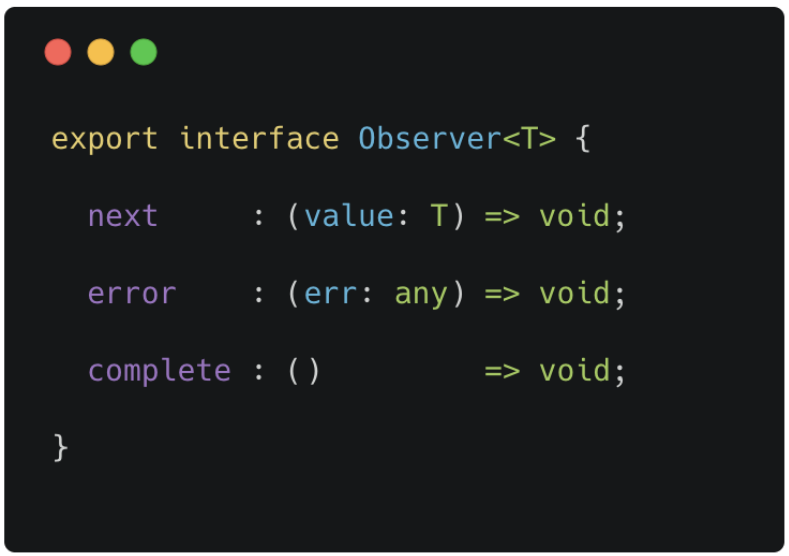
1

```
const observer = {
   next    : console.log,
   error   : console.error,
   complete: console.log
};
const subscriptionFn = (observer) => {

    // Producer activity
    // Annouce activty using observer
    // observer,next(<value>);

    return () => {
        // Stop producer activity (optional)
        // cancels connection to the producer
    };
}

const source$        = new Observable( subscriptionFn );
const subscription   = source$.subscribe( observer );

subscription.unsubscribe();
```

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

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

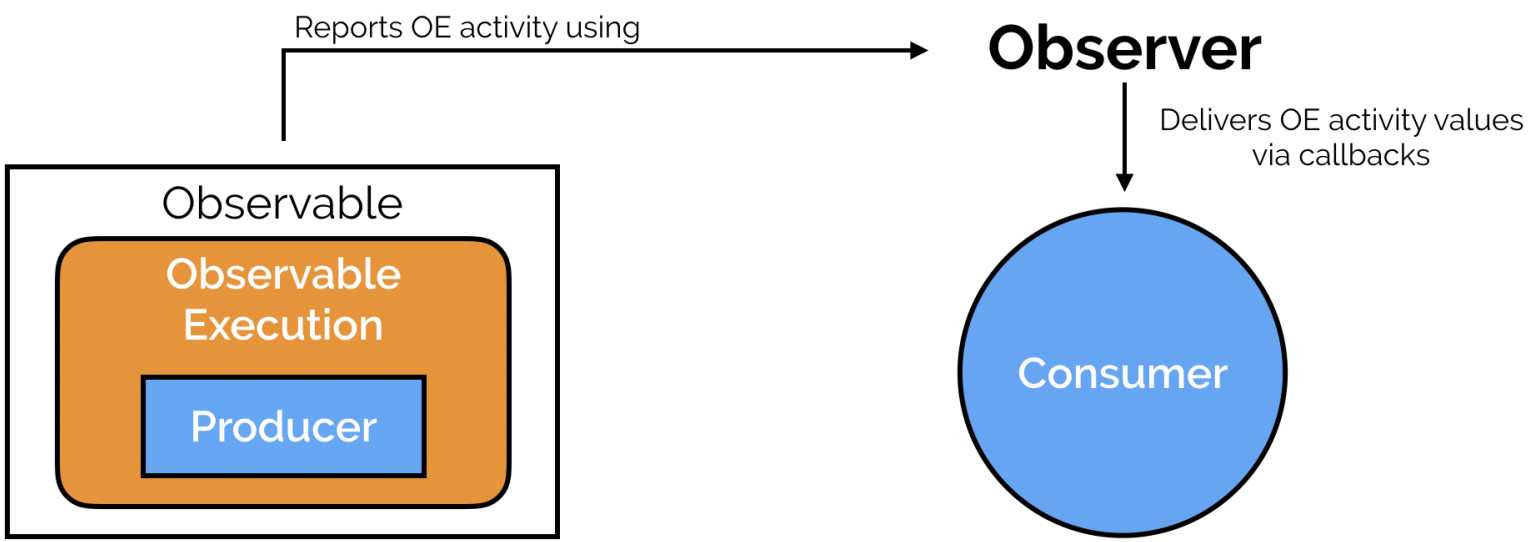
An observable execution is a lazy computation that happen
for EACH Observer that subscribes.

```
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 could be async calls using axios or fetch
* Consumers are the subscriptions to those, where we get pushed the values and react to them

Simple **Custom** **Http** Observables

1

```
// Simple implementation of HttpClient Observable
function http(url) {

  const subscriptionFn = (observer) => {
    const xhr = new XMLHttpRequest();

    xhr.addEventListener('load', () => {
       if( isReady(xhr) ) {
         observer.next(JSON.parse(xhr.responseText));
         observer.complete();
       }
     });

     xhr.open('GET', url);
     xhr.send();

     return () => xhr.abort();
  }

  const response$ = new Observable( subscriptionFn );
  return response$;
}
```

Observables using Async/Await + **fetch**()

1

```
async function doFetch(url, headers = null) {
  const result = await fetch(url, headers);
  const data = await result.json();
  return data;
}

function http(url): Observable {
  const request$ = new Observable(subscriber => {
    let shouldNotify = true;

    doFetch(url).then(list => {
      if (shouldNotify) {
        subscriber.next((contacts = assignUIDs(list)));
        subscriber.complete();
       }
     })
    .catch(subscriber.error);

    return () => {
      shouldNotify = false;
    };
  });
```

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 React

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

2

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

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

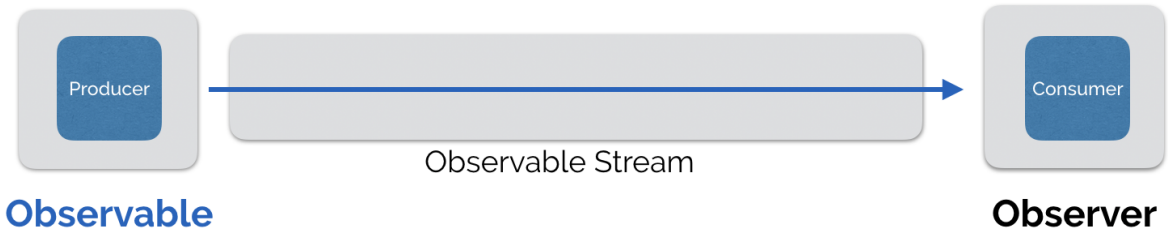
```
export declare class Observable<T>  {

  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 producer = (observer) => {
  observer.next(1);
  observer.complete();
}
const source$ = new Observable(producer);
```

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

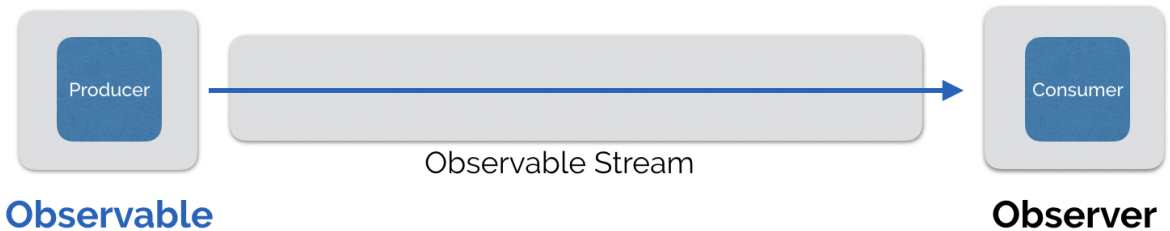
```
// Only watch for emitted values

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




```
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

2

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

Custom Observables (depends)

3rd-Party Observables (depends)

Unsubscribe **Not Needed**

Fetch-based Observable

Some operators; eg. **take**

When using the **useObservable** hook!

Do not assume that the stream auto-completes...
unsubscribe!

Speaker notes

Unsubscribe
Control, Group and Array
(depends) if that observable execution completes
Will talk about operators later
Now the cool thing about the observable contract is that we can use that API to figure out what an observable does.
Include a complete callback function and see if the observable completes after you subscribe. Navigate away from

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
);
```

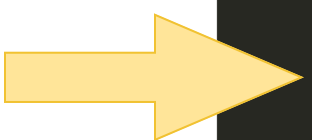
In this case ^, the **Observable Execution** will be the actual internal call to the remote REST endpoint

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 Http 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 Http, each call starts its own observable execution
* Two subscribes can equal two network requests

Coding with **Observables**

2



```
export const TicketsList: React.FC = () => {
  const [service] = useState<TicketsService>(() => new TicketsService());
  const [tickets, setTickets] = useState<Tickets[]>([]);


  useEffect(() => {

    const allTickets$ = service.getTickets();
    allTickets$.subscribe(list => setTickets(list));

  }, [service]);

  return (
    <IonList>
      {tickets.map((ticket, idx) => {
        return <TicketListItem key={idx} ticket={ticket} />;
      })}
    </IonList>
  );
};
```

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:

* No cancellations
* No cleanup on unmount
* Zombies

Traditional **Unsubscribe**()

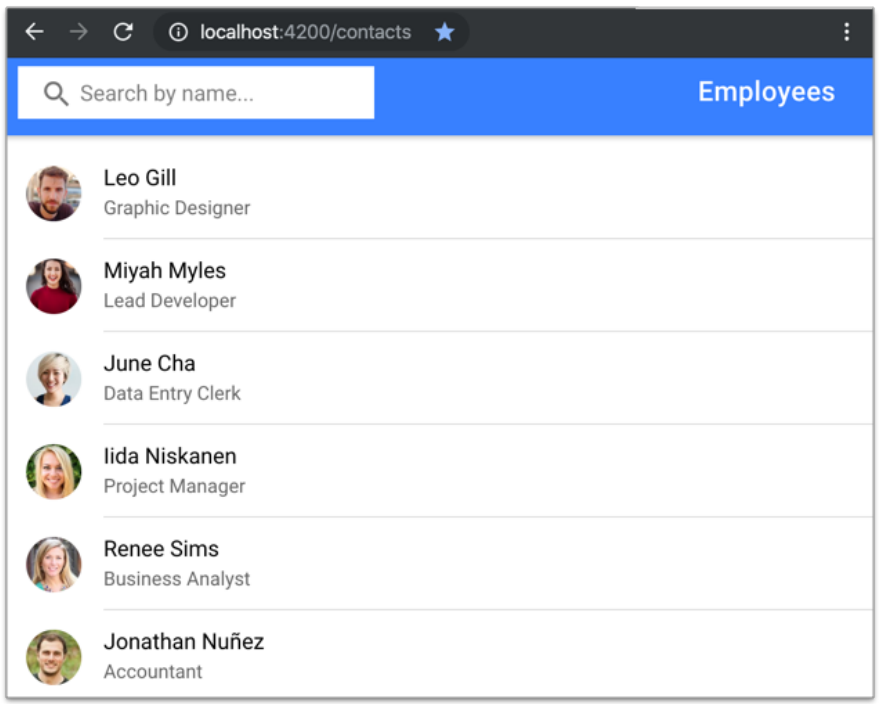
2

```
export const TicketsList: React.FC = () => {
  const [service] = useState<TicketsService>(() => new TicketsService());
  const [tickets, setTickets] = useState<Tickets[]>([]);


  useEffect(() => {

    const allTickets$ = service.getTickets();
    const connection = allTickets$.subscribe(list => setTickets(list));

    return () => connection.unsubscribe();
  }, [service]);

  return (
    <IonList>
      {tickets.map((ticket, idx) => {
        return <TicketListItem key={idx} ticket={ticket} />;
      })}
    </IonList>
  );
};
```

This is a typical solution seen in the community.

Later... we will use a better approach!

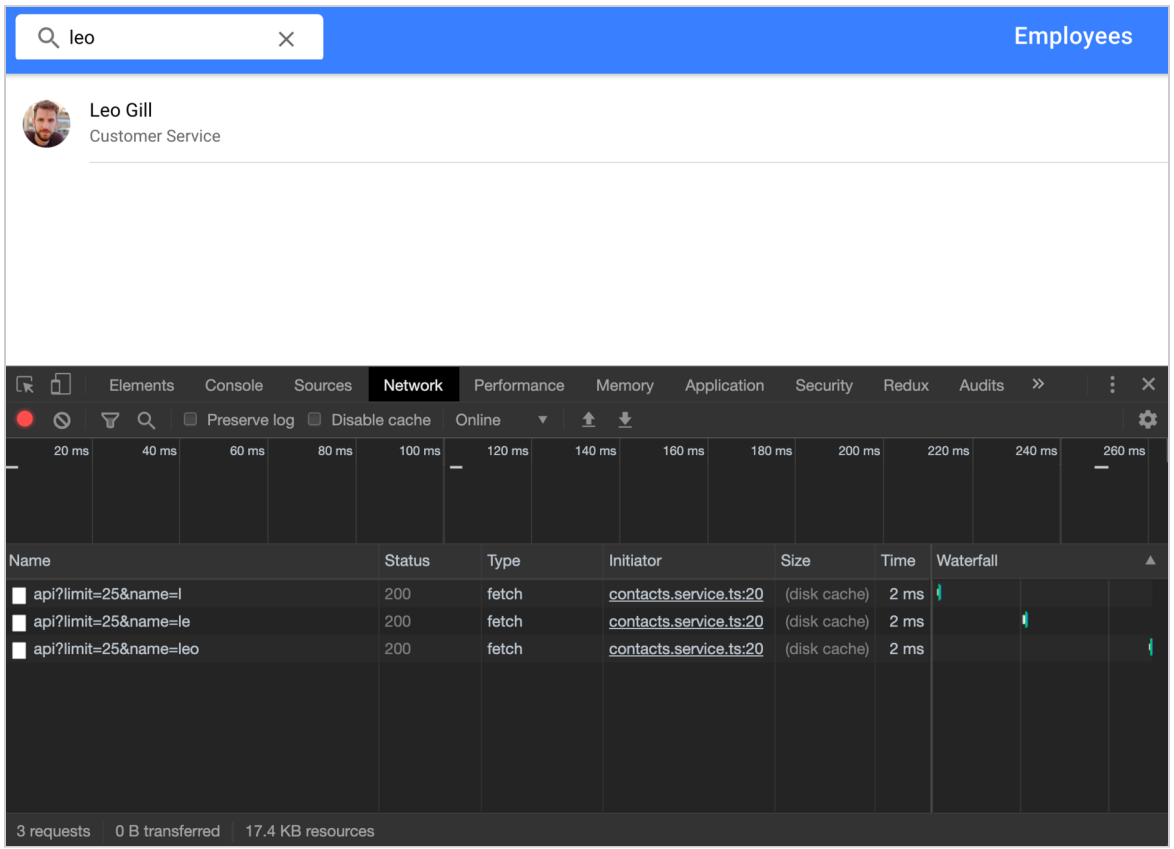
RxJS Lab 1: Use Observables



- Convert use of Promises to **Observables**
- Use Observable **subscribe()** to start network request
- Use `**setPeople()**` to update list when the stream emits

Lab Exercise

Problems with Basic Search

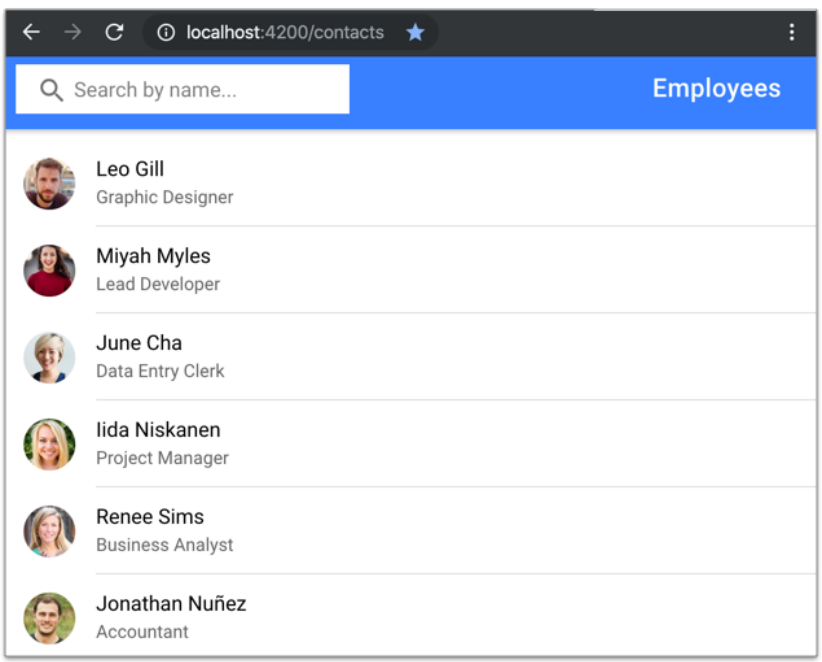


This simple search implementation comes with (3) problems:

- We hit the server on every keystroke
- We hit the server unnecessarily
- You tell us! What is the last issue ?

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 use:

- **RxJS 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

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**:

Example *projection functions* to transform or block:

(i) => i % 2

(i) => i.toUpperCase()

```
[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 **Operators**

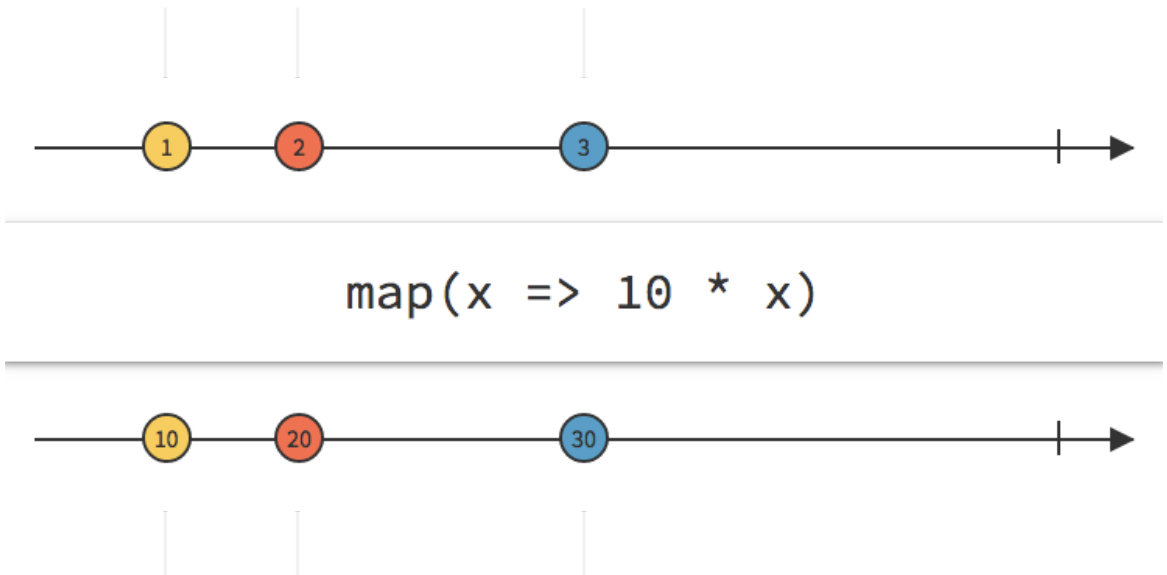
- Are pure functions
- **Input read-only** observable
- **Output new** observable (generated)

Why ?

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

RxJS **Operators**

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



Operators allow custom projection functions to project *or block* each value in the stream into another value

Custom map() RxJS Operator

3

fn is called the projection function

```
function map(fn) {
  return (incoming$) => {

    const outgoing$ = new Observable(subscriptionFn);

    const subscriptionFn = (observer) => {
        const subscription = source.subscribe(
          (value) => observer.next( fn(value) );
        );

        return () => subscription.unsubscribe();
    };

    return outgoing$;

  }
}
```

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

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

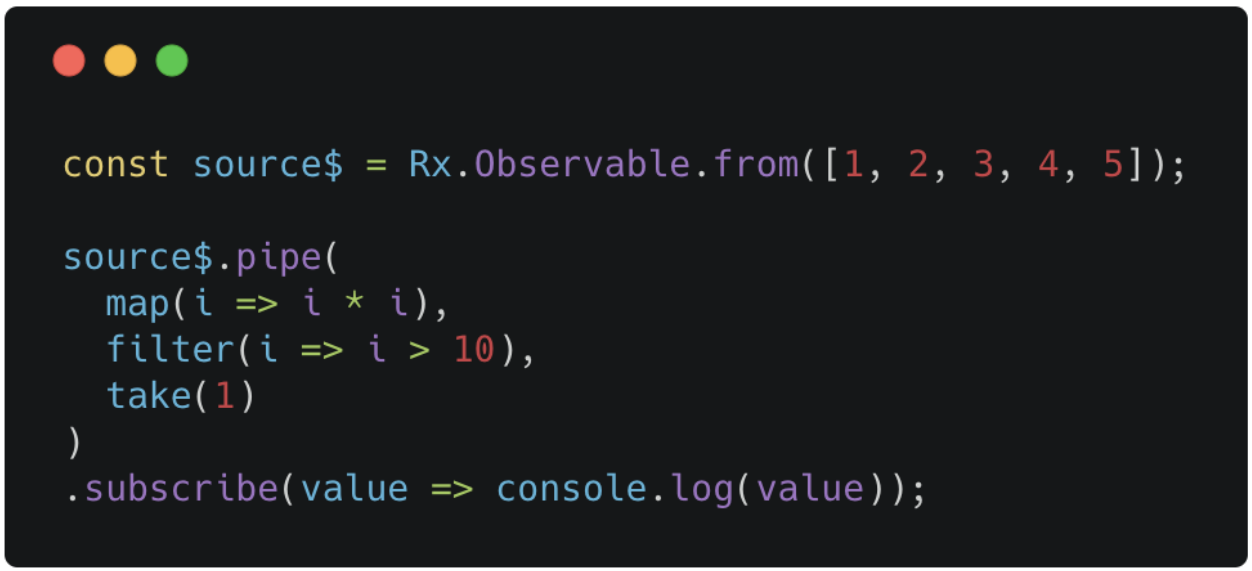
Speaker notes

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

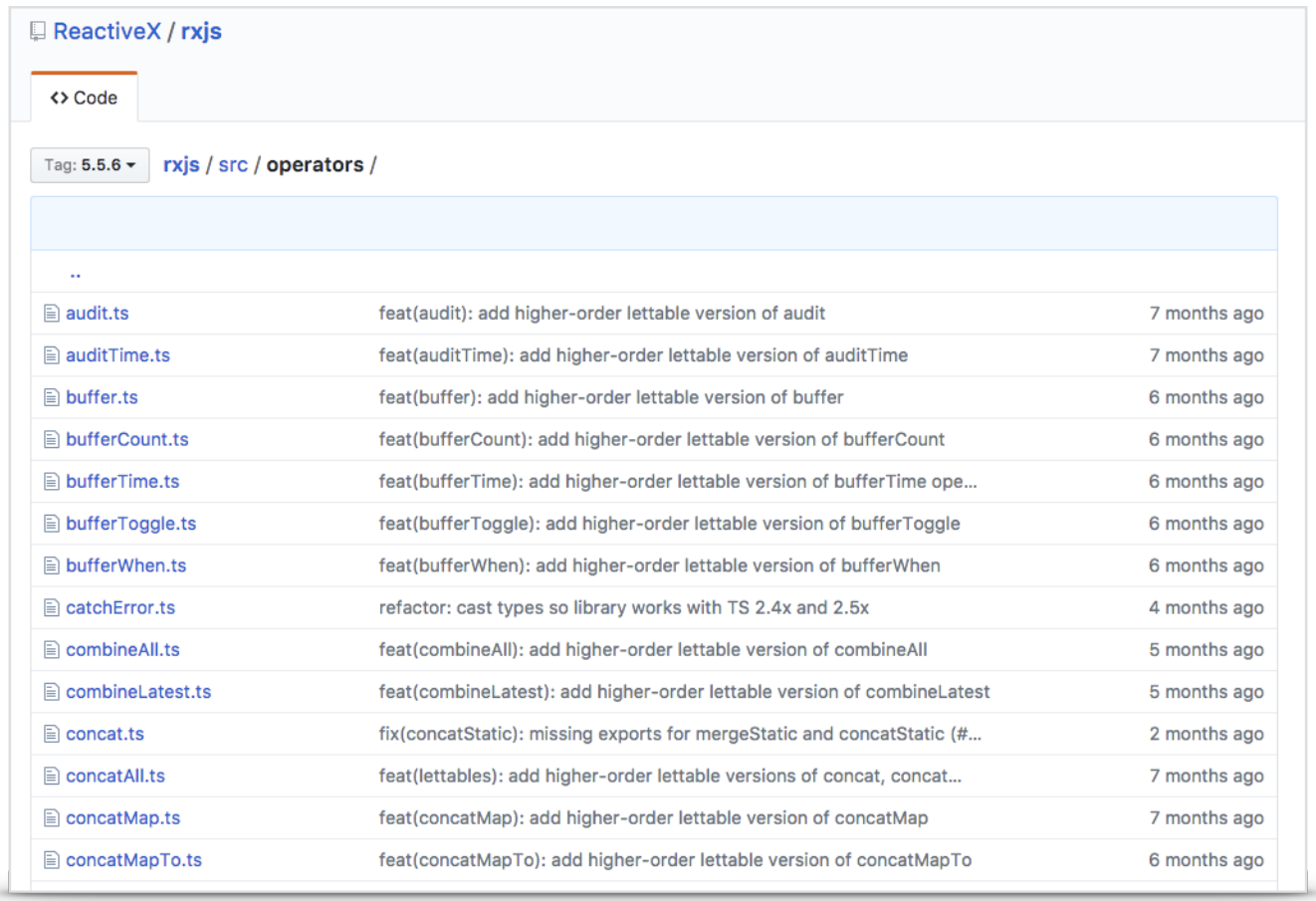


```
const source$ = Rx.Observable.from([1, 2, 3, 4, 5]);

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**

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

RxJS comes with a bunch of **Operators**



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...

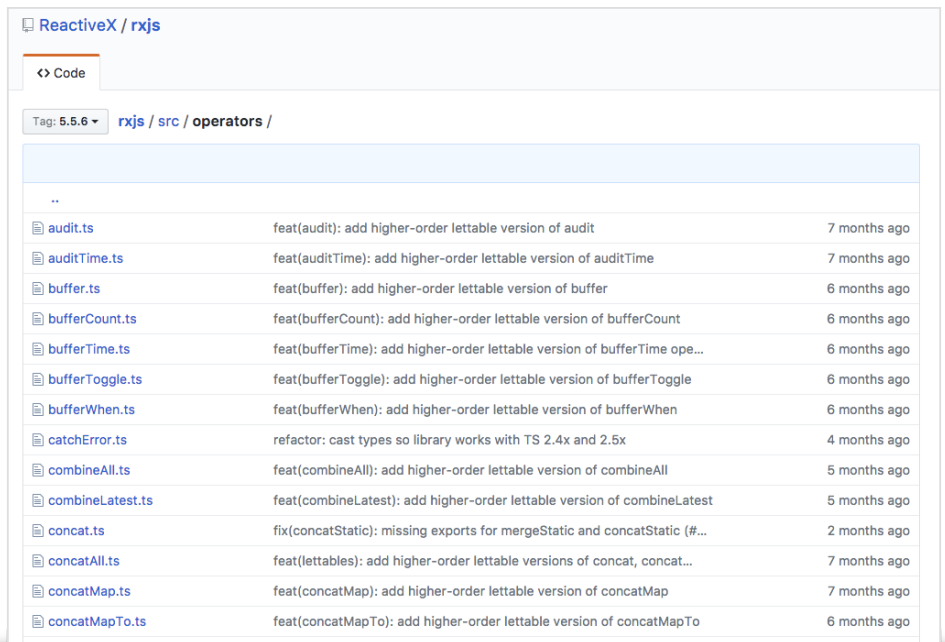
https://rxjs-dev.firebaseapp.com/guide/operators

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...

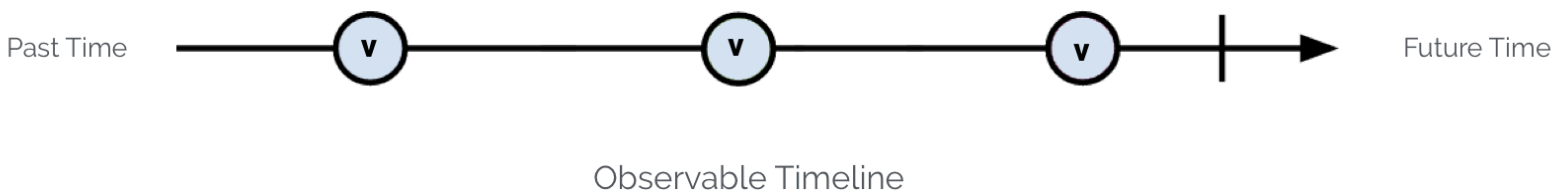
Developers really only need to know **4 - 6 operators** to get started!

Marble Diagrams

Diagrams help visualize Observable events and RxJS operators!

RxMarbles.com

Marble Diagrams



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)?

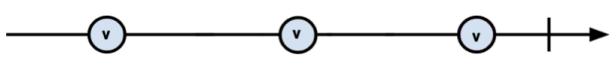
```
1,2,3      #          |
```

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


```
const source$ = new Observable( observer => {

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

  observer.complete();

});
```



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

Speaker notes

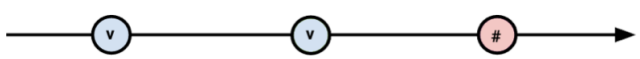
Three values then complete

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

Observable Visualizations

3

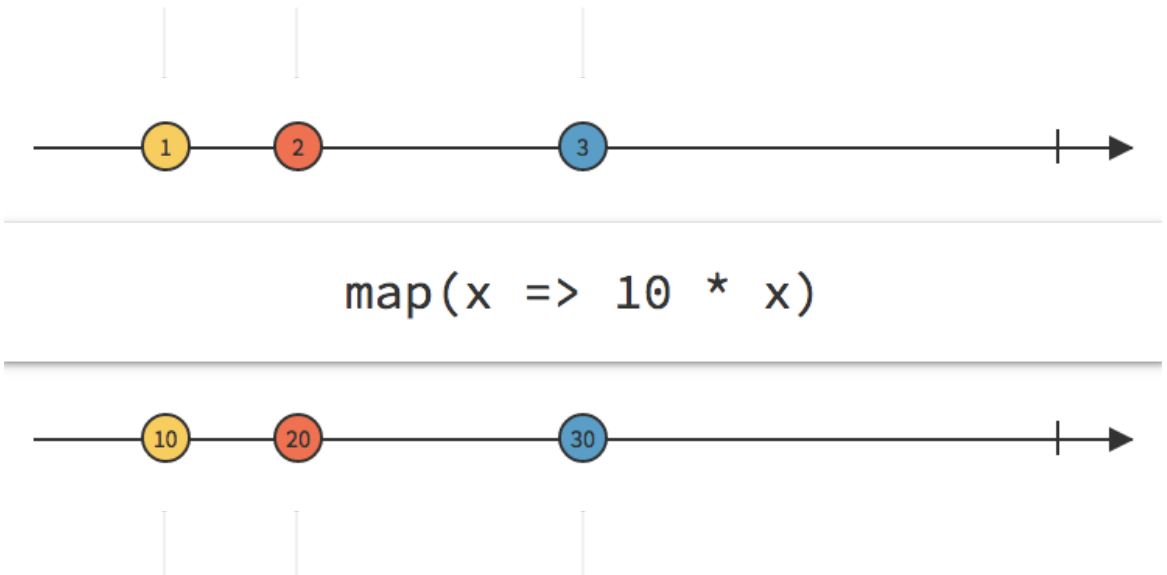
```
const source$ = new Observable( observer => {

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


  observer.error( "ooops!" );

});
```




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

Map Operator

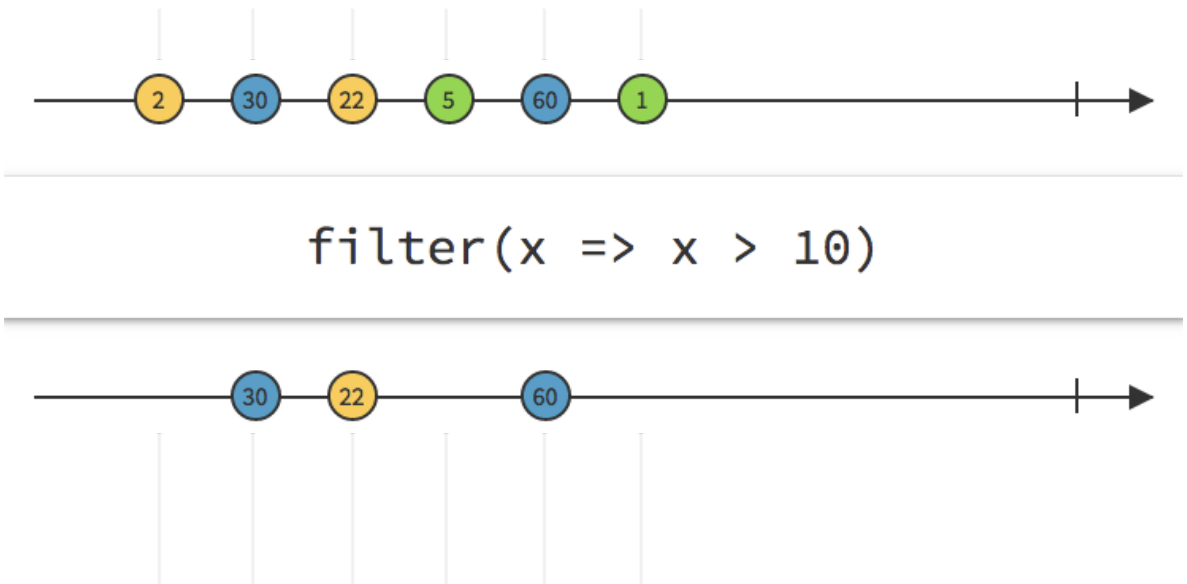
Transform the items emitted by an Observable by applying a function to each item



StackBlitz Demo

Filter Operator

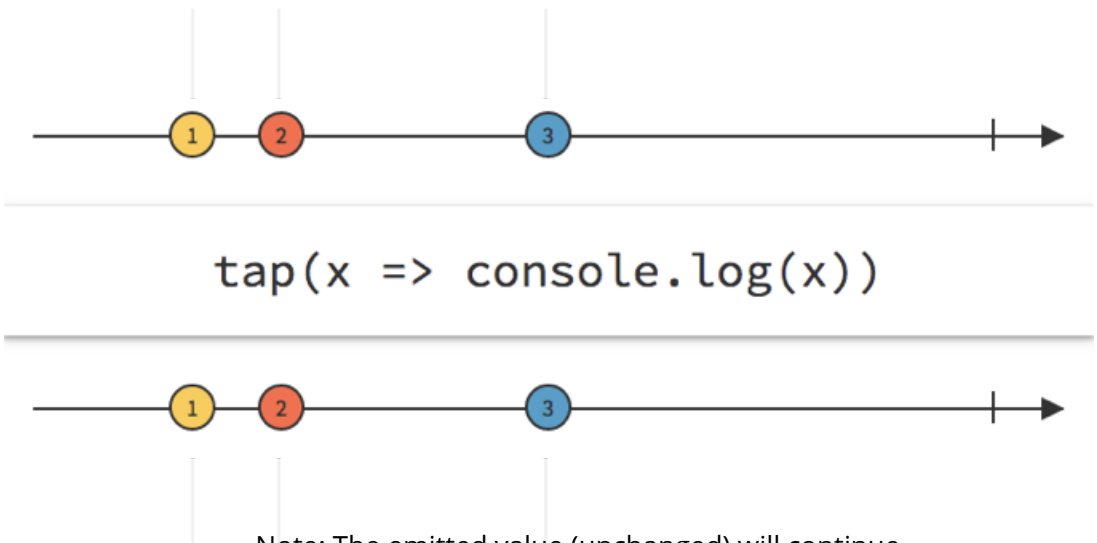
Emit only those Observable stream values that pass a predicate test



StackBlitz Demo

Tap Operator

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

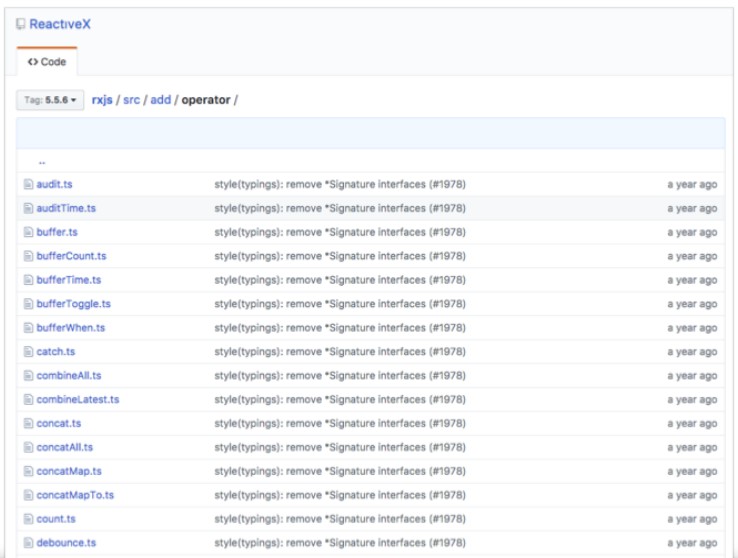


Note: The emitted value (unchanged) will continue to propagate through the pipe.

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


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

import { map, filter } from '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

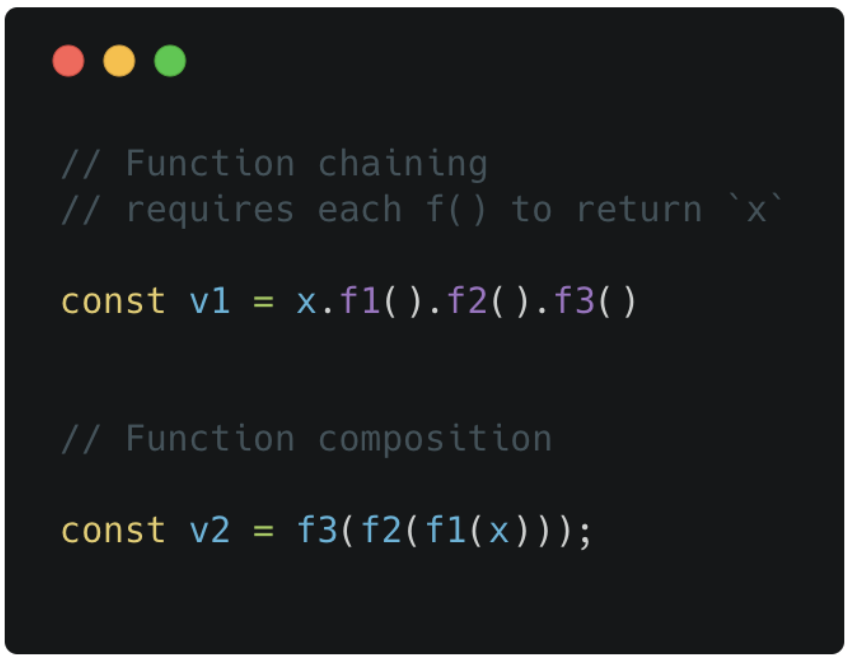
If it's outside the pipe,
import from 'rxjs'

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...

3

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

```
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

Problem #1: Server Thrashing


```
const subscription = userSearch$.subscribe( criteria => {
   const request$ = service.searchBy(criteria);

   request$.subscribe(contacts => setPeople(contacts));
});
```

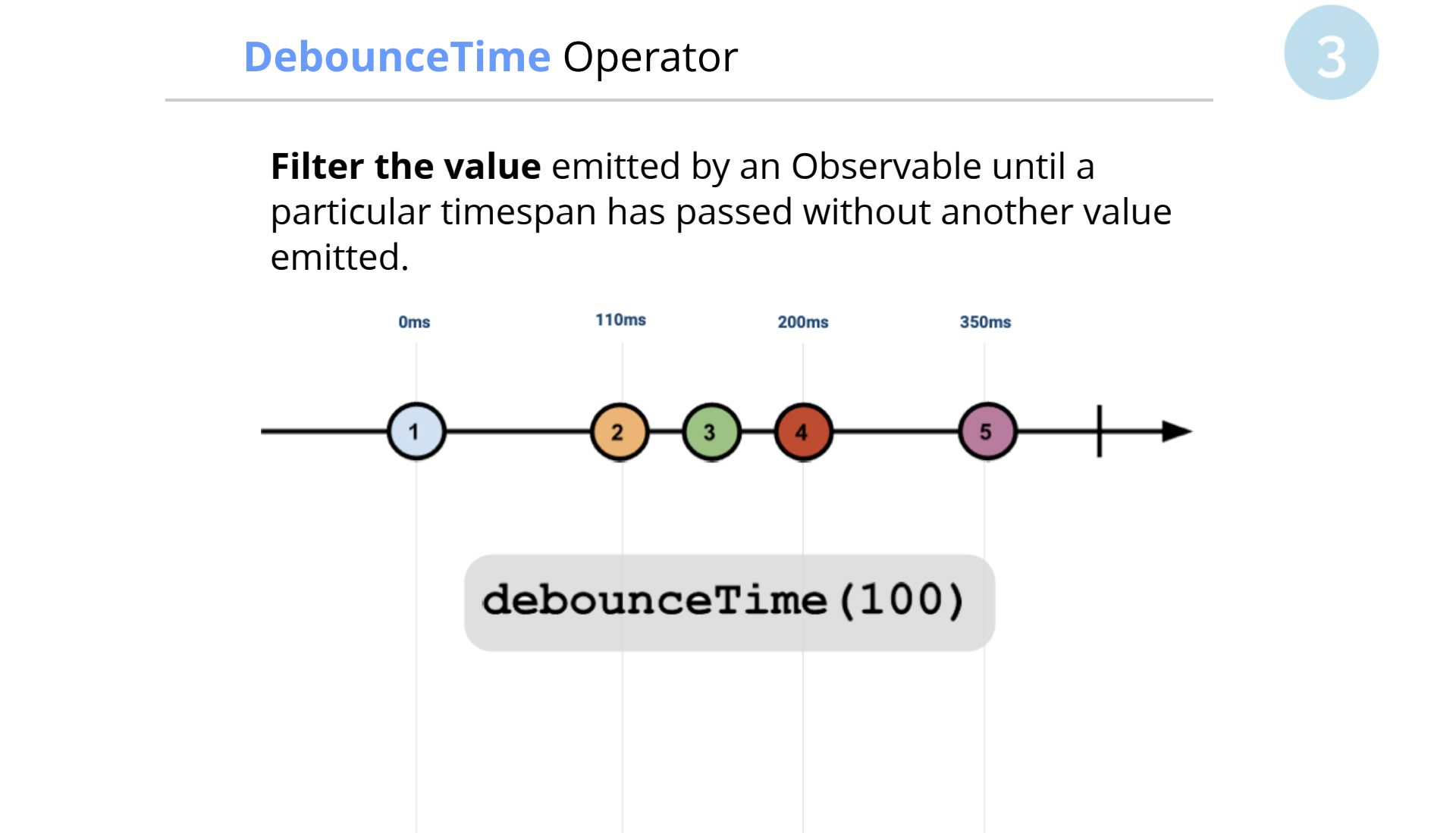
Problem: code calls to the server EVERY keystroke

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

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

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 RxJS Operator: **debounceTime()**

3

```
const controlledSearch$ = userSearch$.pipe(
  debounceTime(250)
);

const subscription = controlledSearch$.subscribe( criteria => {
   const request$ = service.searchBy(criteria);

   request$.subscribe(contacts => setPeople(contacts));
});
```

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

Problem #2: Server Thrashing

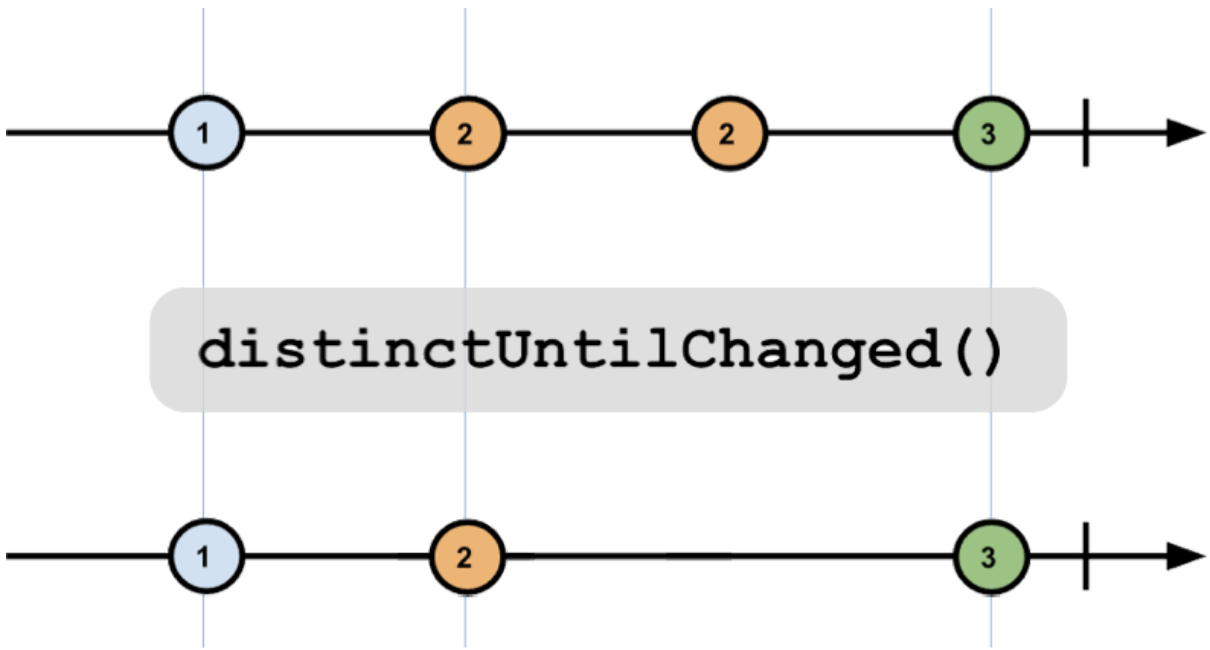
3

```
const controlledSearch$ = userSearch$.pipe(
  debounceTime(250)
);

const subscription = controlledSearch$.subscribe( criteria => {
   const request$ = service.searchBy(criteria);

   request$.subscribe(contacts => setPeople(contacts));
});
```

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

DistinctUntilChanged Operator

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 RxJS Operator: **distinctUntilChanged()**

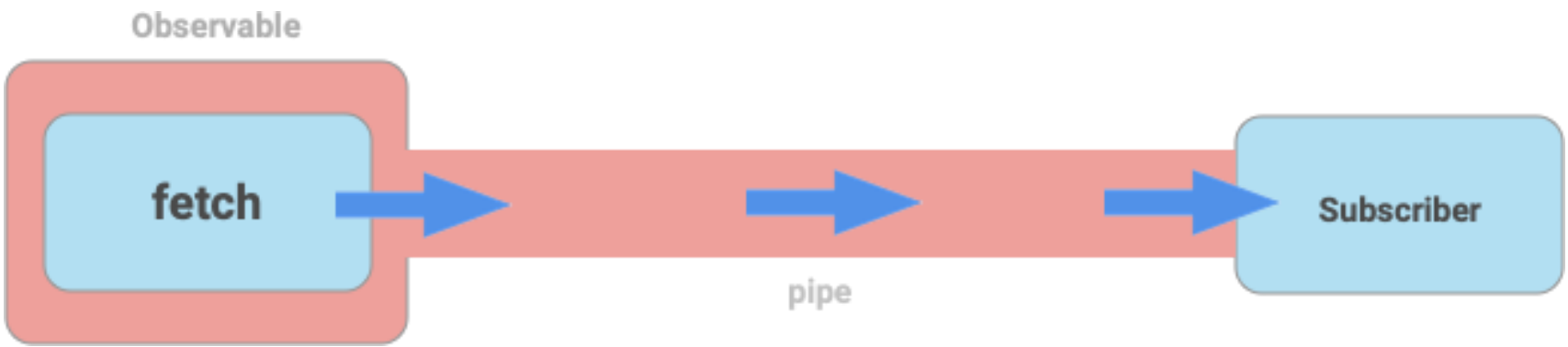

```
const controlledSearch$ = userSearch$.pipe(
  debounceTime(250),
  distinctUntilChanged()
);

const subscription = controlledSearch$.subscribe( criteria => {
   const request$ = service.searchBy(criteria);

   request$.subscribe(contacts => setPeople(contacts));
});
```

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

Streams of data to Observers



Streams are read-only data emitters

Producer **fetch** is internal and private

This system is closed to the outside world.

How to can **we inject data** to be emitted?

Speaker notes

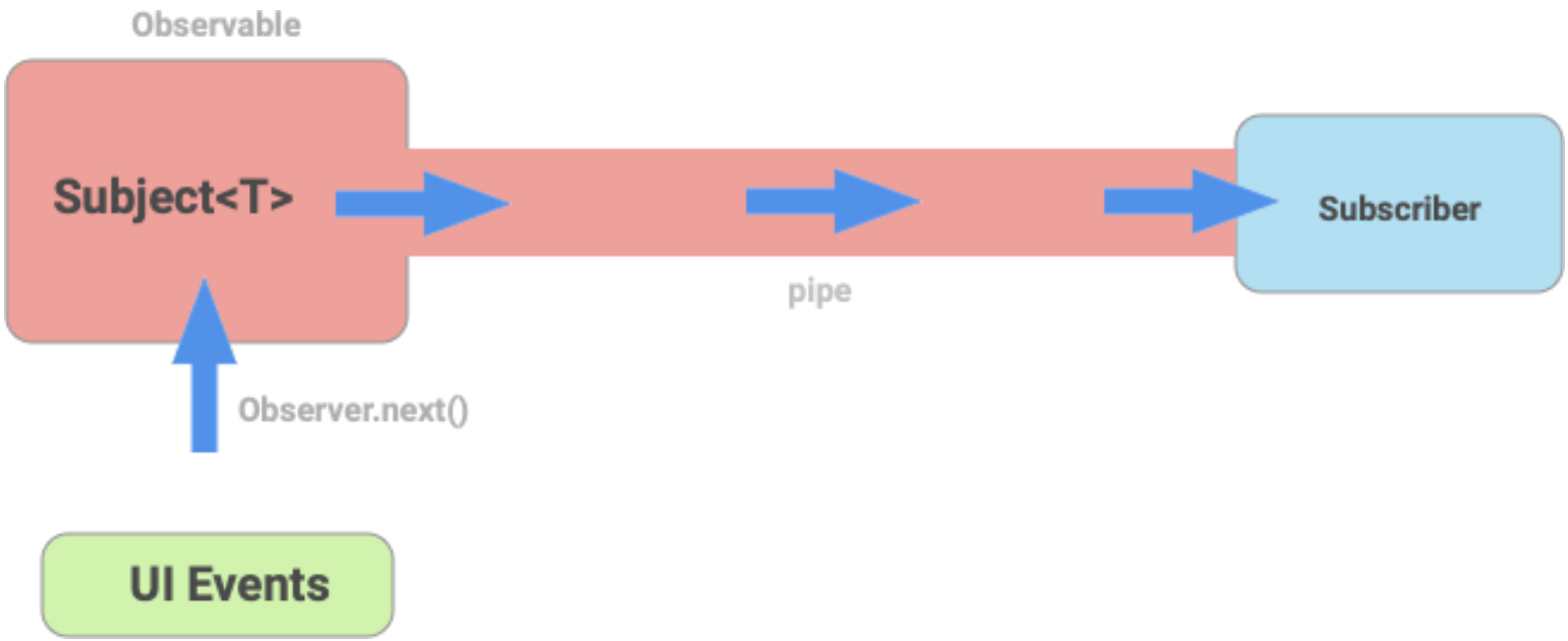
To use the power of RxJS operators, we need observables.
How do we listen of input events and send those through an observable stream?

RxJS Streams for UI Events


```
<IonItem style={inlineItem}>
  <IonIcon name="search"></IonIcon>

  <IonInput  autofocus style={iconOnLeft} type="text"
             placeholder="Search by name..."
             ref={searchByName}>
  </IonInput>

</IonItem>
```

How do we **consume** UI DOM Events as a **RxJS stream**?

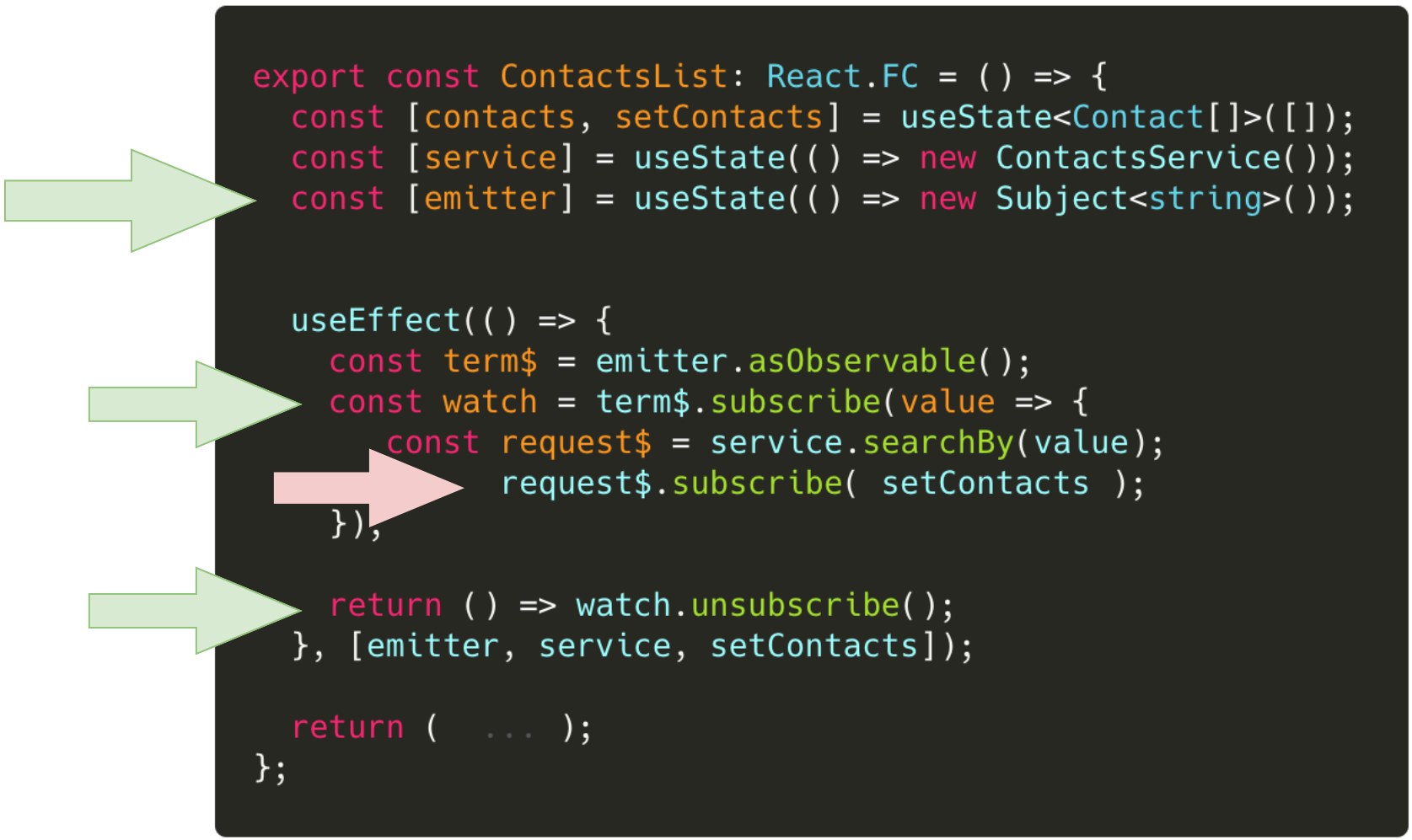
RxJS Subject<T>



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

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

RxJS Subject<T>



```
export const ContactsList: React.FC = () => {
  const [contacts, setContacts] = useState<Contact[]>([]);
  const [service] = useState(() => new ContactsService());
  const [emitter] = useState(() => new Subject<string>());


  useEffect(() => {
    const term$ = emitter.asObservable();
    const watch = term$.subscribe(value => {
       const request$ = service.searchBy(value);
             request$.subscribe( setContacts );
    });

    return () => watch.unsubscribe();
  }, [emitter, service, setContacts]);

  return (   ...  );
};
```

contacts-list.tsx

Speaker notes

A Subject is an emitter and an Observable
Here in our imperative code, we prepare a stream to receive `search terms` and call the HttpService/REST server
What is term$....
We have a memory leak here... can you tell where ?

RxJS Subject<T>

3

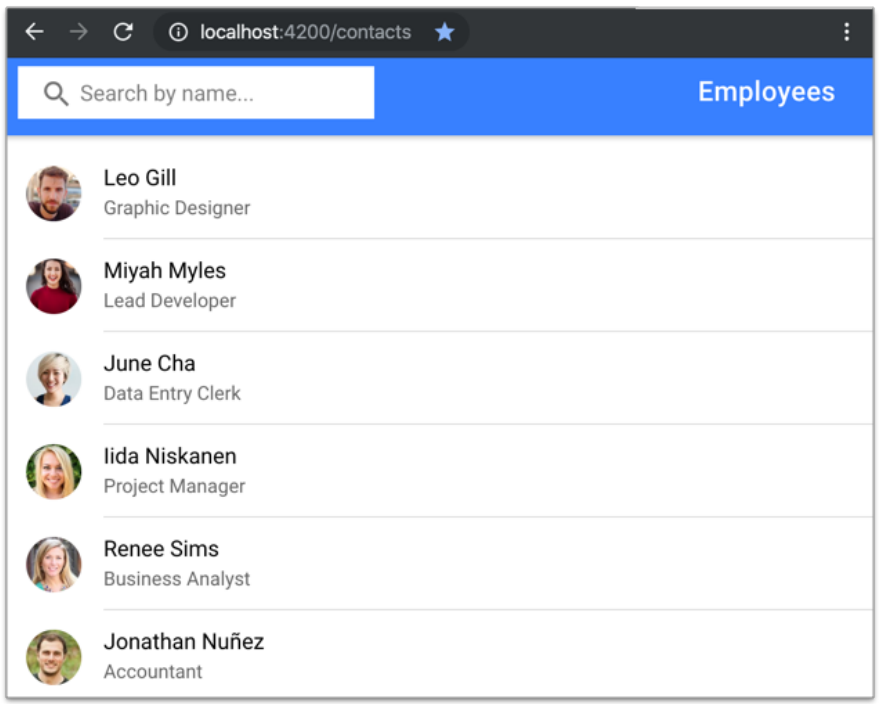
```
export const ContactsList: React.FC = () => {
  const [contacts, setContacts] = useState<Contact[]>([]);
  const [service] = useState(() => new ContactsService());
  const [emitter] = useState(() => new Subject<string>());

  useEffect(() => {
    const term$ = emitter.asObservable();
    const watch = term$.subscribe(value => {
         const request$ = service.searchBy(value);
         request$.subscribe( setContacts );
    });

    return () => watch.unsubscribe();
  }, [emitter, service, setContacts]);

  return (
    <IonInput placeholder="Search by name..."
              onIonChange={(e) =>  emitter.next(e.target.value)} >
    </IonInput>

  );
};
```

contacts-list.tsx

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.
Discuss difference between useState() and subject.next()
Why this complexity to push input value thru a stream ?

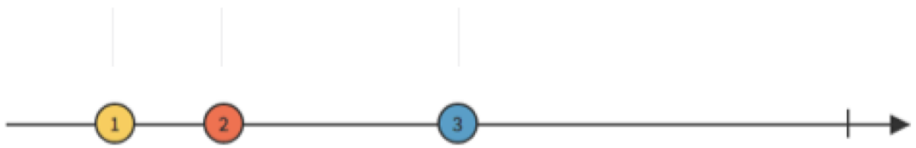
RxJS Lab 2: Throttle Search Requests



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

Lab Exercise

Types coming through Observable streams

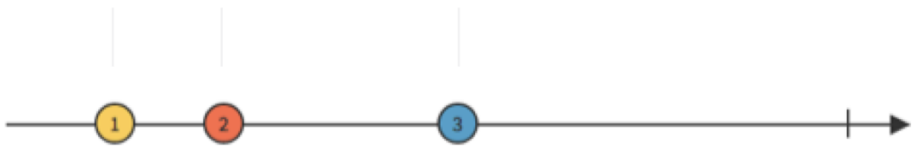


An observable may emit value of **any** JavaScript type:

- 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

```
export type InputEmitter = Subject<string>;

export const ContactsList: React.FC = () => {
  const [service] = useState<ContactsService>(() => new ContactsService());
  const [emitter] = useState<InputEmitter>(() => new Subject<string>());
  const [contacts, setContacts] = useState<Contact[]>([]);

    useEffect(() => {
      const userSearch$ = emitter.asObservable().pipe(
        debounceTime(250),
        distinctUntilChanged(),
        map( criteria => service.searchBy(criteria) )
      );

      userSearch$.subscribe( setContacts );
    }, [emitter, service, setContacts]);

    return (
      <IonInput autofocus
                placeholder="Search by name..."
                onIonChange={ e => emitter.next(e.target.value) }>
      </IonInput>
    );
}
```

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

Speaker notes

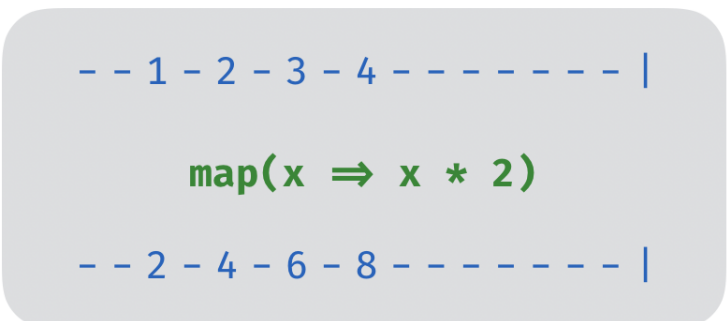
So you could imagine, and maybe you have got into this situation (I certainly did a lot)…
We can use this pipe and these operators to set up the flow of our observable work to be catered towards what we need, surely we could do something there to not have to subscribe to multiple streams? Let's just 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

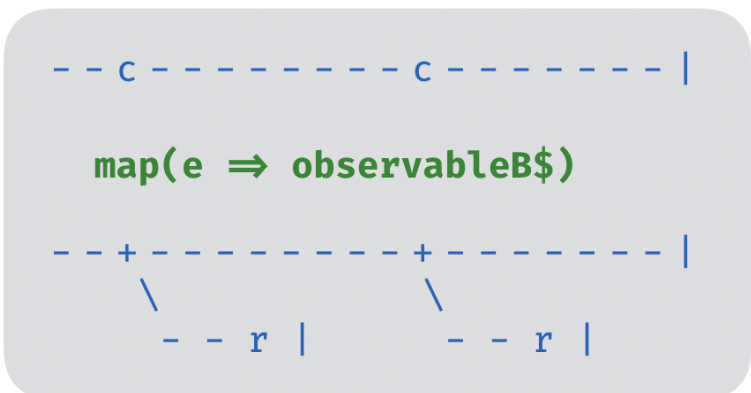
1st Order

- 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

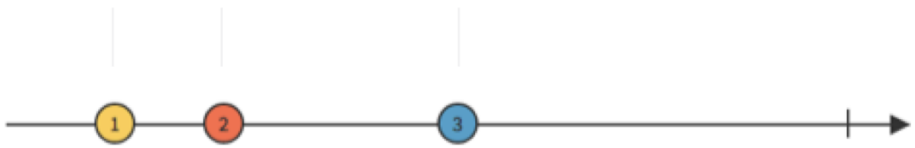


+ is an inner observable

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

```
export type InputEmitter = Subject<string>;

export const ContactsList: React.FC = () => {
  const [service] = useState<ContactsService>(() => new ContactsService());
  const [emitter] = useState<InputEmitter>(() => new Subject<string>());
  const [contacts, setContacts] = useState<Contact[]>([]);

    useEffect(() => {
      const userSearch$ = emitter.asObservable().pipe(
        debounceTime(250),
        distinctUntilChanged(),
        map( criteria => service.searchBy(criteria) )
      );

      userSearch$.subscribe( setContacts );
    }, [emitter, service, setContacts]);

    return (
      <IonInput autofocus
                placeholder="Search by name..."
                onIonChange={ e => emitter.next(e.target.value) }>
      </IonInput>
    );
}
```

Speaker notes

Here we subscribe to manually extract the raw data
But this does not work because map() returns an Observable.

Bad Solution:

Use Nested Subscribes

```
export const ContactsList: React.FC = () => {
    const [service] = useState<ContactsService>(new ContactsService());
    const [emitter] = useState<Subject<string>>(new Subject<string>());
    const [contacts, setContacts] = useState<Contact[]>([]);

    useEffect(() => {
      const userSearch$ = emitter.asObservable()

      userSearch$.pipe(
        map( criteria => {
          let pending: Subscriptions;
          const cancelPending = () => {
            pending && pending.unsubscribe();
            pending = null;
          };

          cancelPending();

          return new Observable( subscriber => {

            try {
               const request$ = service.searchBy(criteria);
               pending = request$.subscribe(list => {
                 subscriber.next(list);
                 subscriber.complete();
               });
            } catch(e) {
              subscriber.error(e);
            }

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

        })
      );

      userSearch$.subscribe( setContacts );

    }, [service, emitter, setContacts]);

    return (
      // ....
    );
}
```

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 const ContactsList: React.FC = () => {
    const [service] = useState<ContactsService>(new ContactsService());
    const [emitter] = useState<Subject<string>>(new Subject<string>());
    const [contacts, setContacts] = useState<Contact[]>([]);

    useEffect(() => {
      const userSearch$ = emitter.asObservable()

      userSearch$.pipe(
        map( criteria => {
          let pending: Subscriptions;
          const cancelPending = () => {
            pending && pending.unsubscribe();
            pending = null;
          };

          cancelPending();

          return new Observable( subscriber => {

            try {
               const request$ = service.searchBy(criteria);
               pending = request$.subscribe(list => {
                 subscriber.next(list);
                 subscriber.complete();
               });
            } catch(e) {
              subscriber.error(e);
            }

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

        })
      );

      userSearch$.subscribe( setContacts );

    }, [service, emitter, setContacts]);

    return (
      // ....
    );
}
```

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

```
export type InputEmitter = Subject<string>;

export const ContactsList: React.FC = () => {
  const [contacts, setContacts] = useState<Contact[]>([]);
  const [service] = useState<ContactsService>(() => new ContactsService());
  const [emitter] = useState<InputEmitter>(() => new Subject<string>());

    useEffect(() => {
      const userSearch$ = emitter.asObservable()
      const watch = userSearch$.pipe(
        debounceTime(250),
        distinctUntilChanged(),
        switchMap( criteria => service.searchBy(criteria) )
      ).subscribe( setContacts );

      return () => watch.unsubscribe();

    }, [emitter, service, setContacts] );

    return (
      // ....
    );
}
```

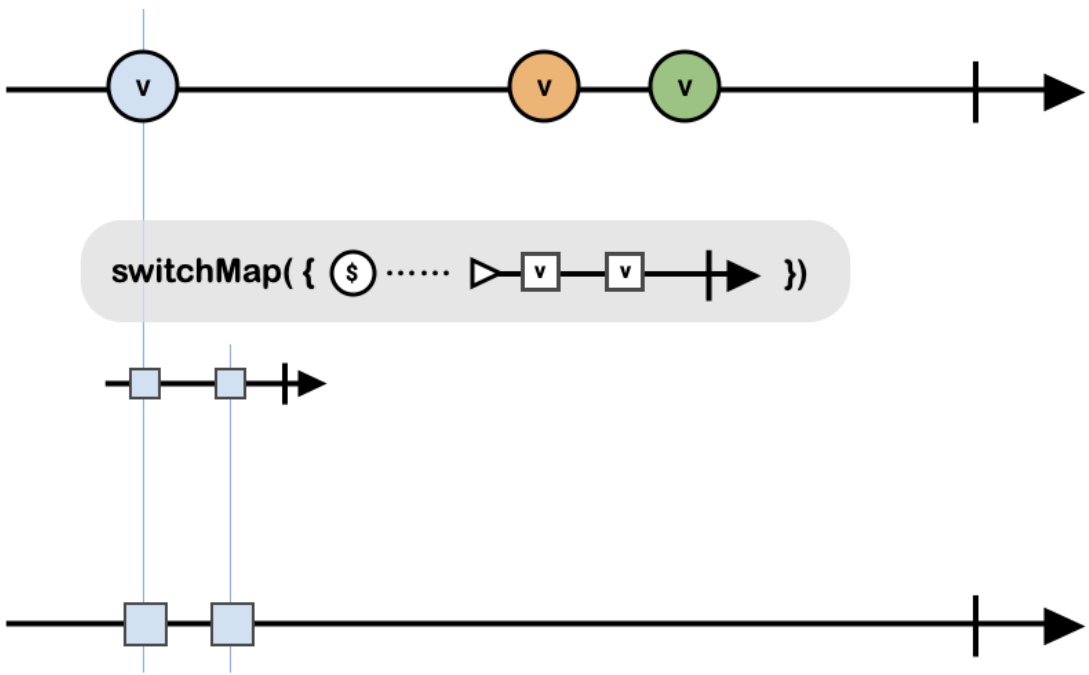
When we get a new value...

switch to watch a nested observable

1, single unsubscribe()

switchMap Operator

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

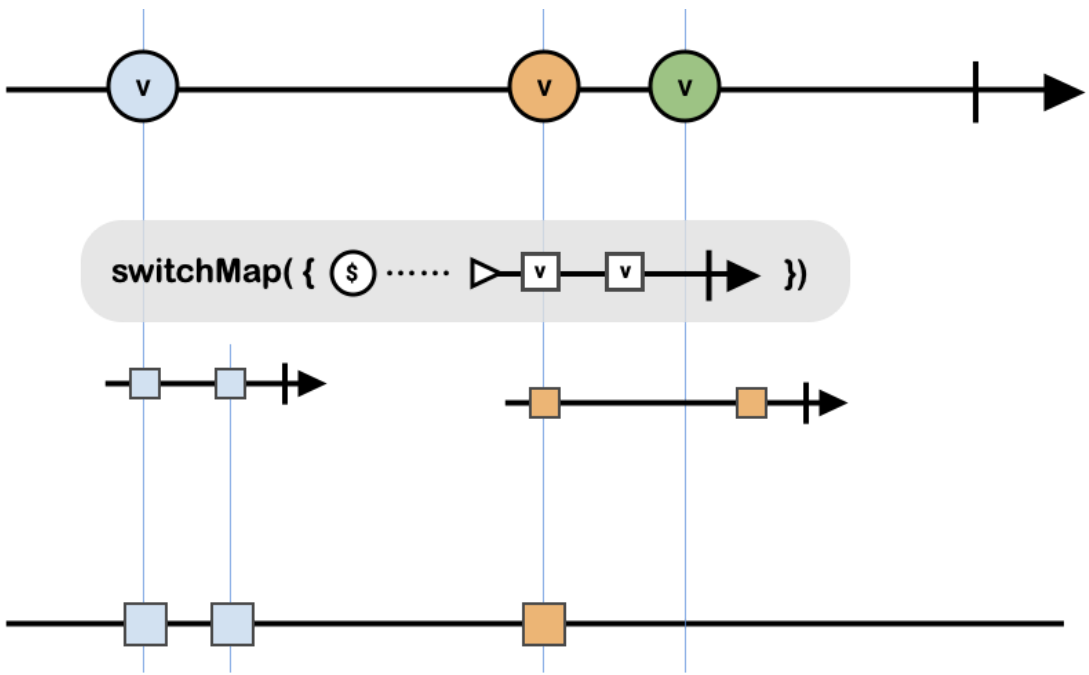


Speaker notes

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.



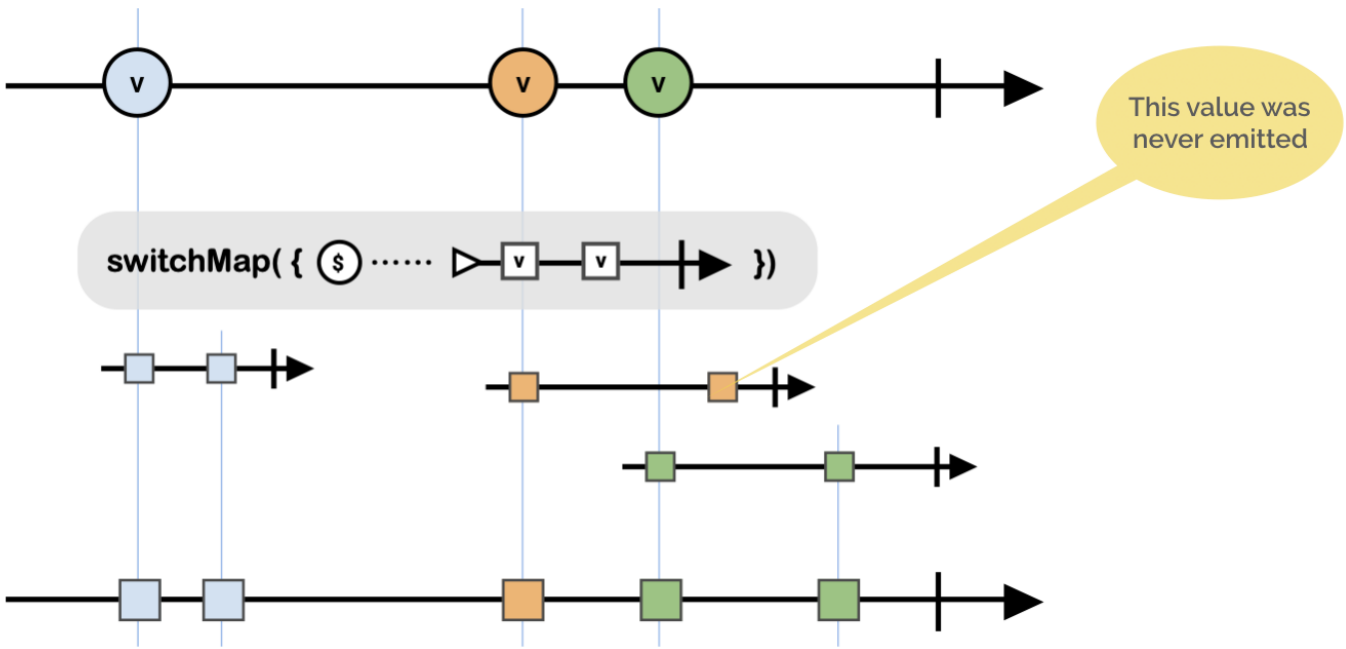
Speaker notes

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

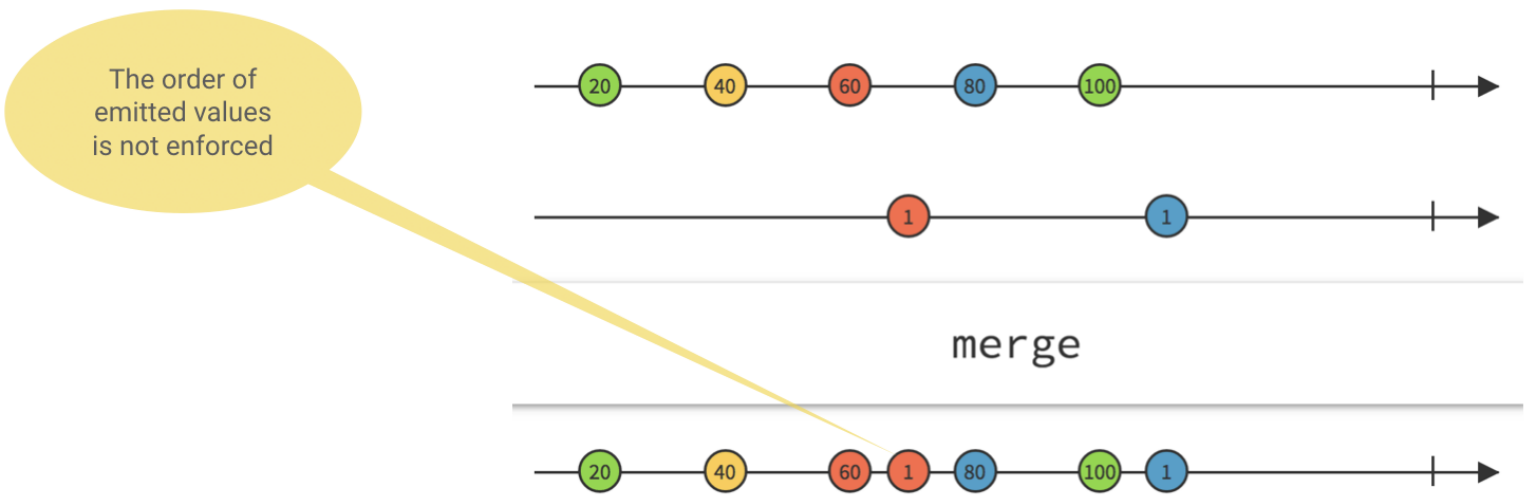
Other common operators

switchMap, mergeMap, concatMap, forkJoin

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

Merge

Combine multiple observables by merging their emissions.



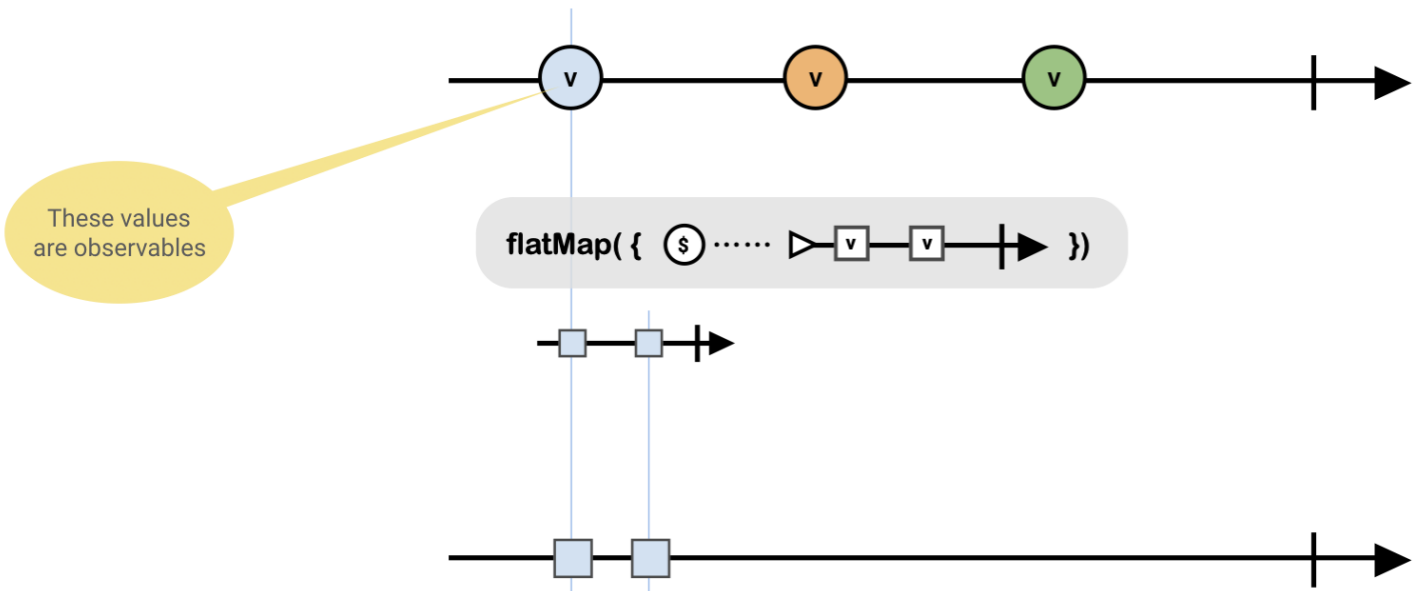
StackBlitz Demo

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.



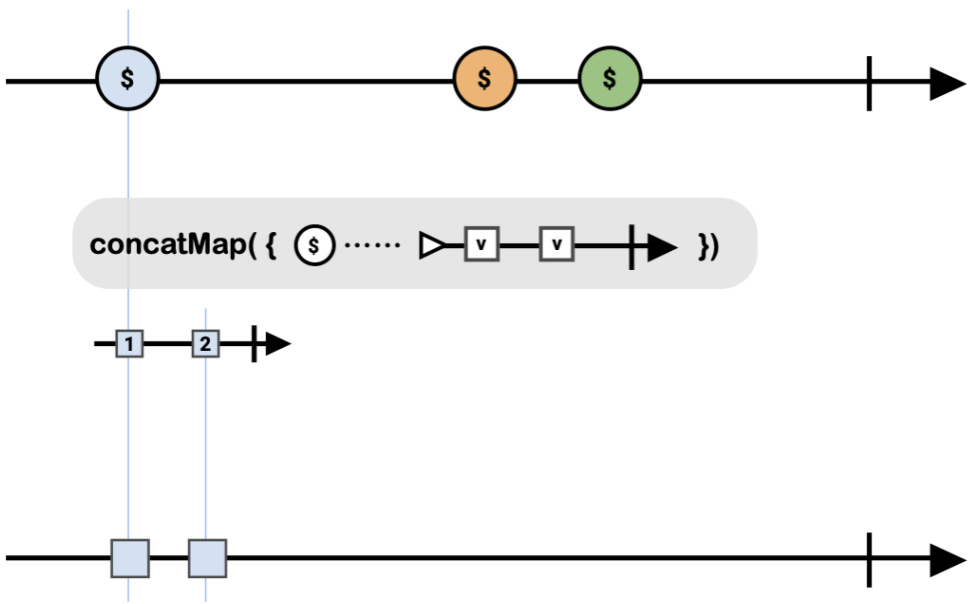
Speaker notes

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.

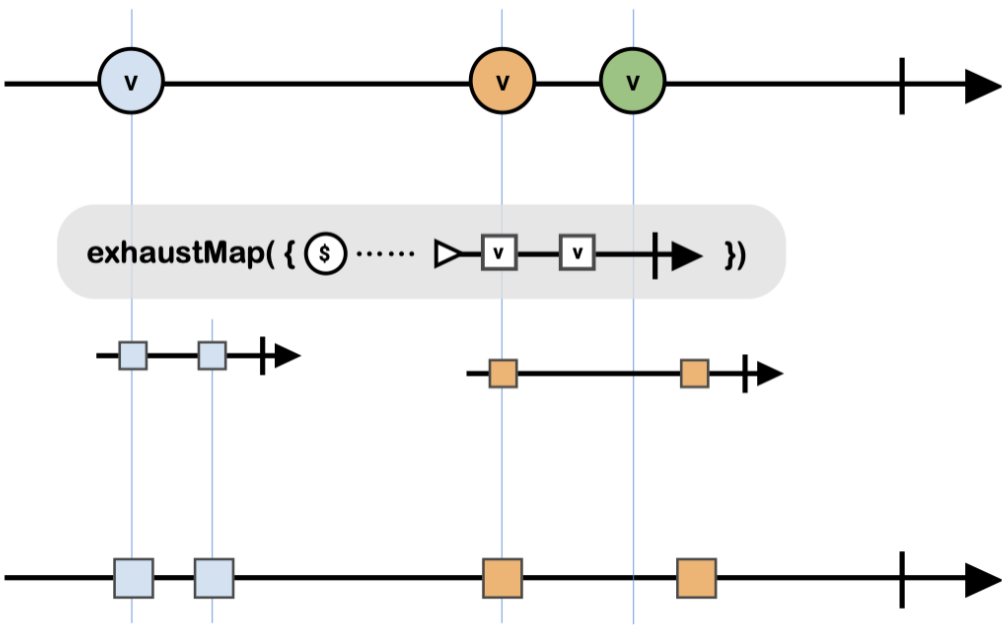


Speaker notes

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.

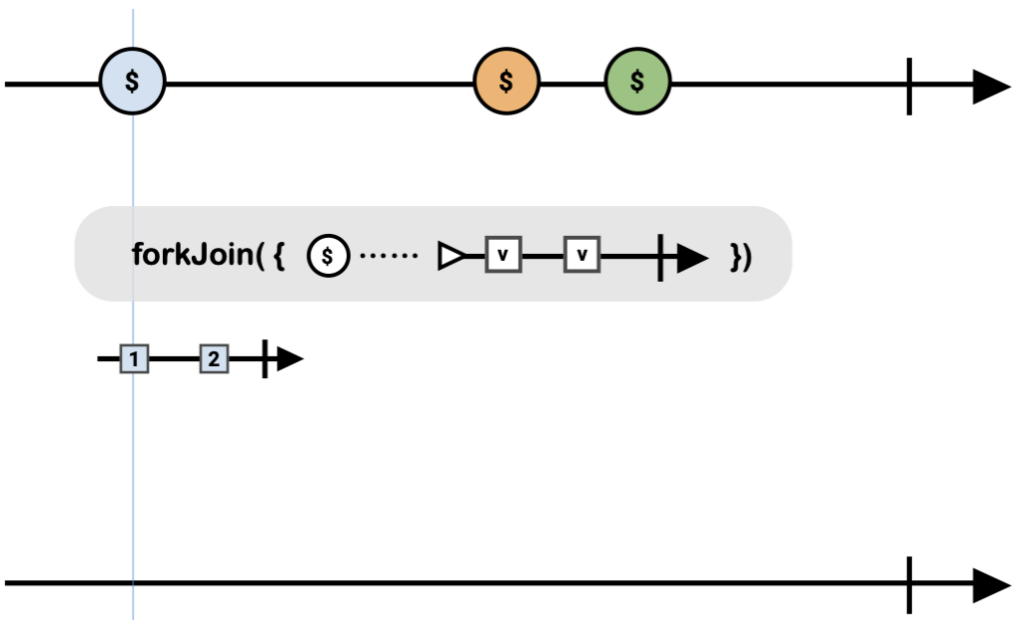
When you need a **Higher-Order** operator

3

- 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.



RxJS Lab 3: Out-of-Order Responses

```
export const ContactsList: React.FC = () => {
  const [contacts, setContacts] = useState<Contact[]>([]);
  const [service] = useState<ContactsService>(() => new ContactsService());
  const [emitter] = useState<InputEmitter>(() => new Subject<string>());

    useEffect(() => {
      const userSearch$ = emitter.asObservable()
      const watch = userSearch$.pipe(
        debounceTime(250),
        distinctUntilChanged(),
        switchMap( criteria => service.searchBy(criteria) )
      ).subscribe( setContacts );
```

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

Lab Exercise

RxJS Lab 4: Refactor to Service

```
export const ContactsList: React.FC = () => {
  const [people, setPeople] = useState<Contact[]>([]);
  const [criteria, setCriteria] = useState<string>('');
  const [service] = useState<ContactsService>(injector.get(ContactsService));
  const [emitter] = useState<Subject<string>>(new Subject<string>());

  const doSearch = (e: Event) => {
    const term = (e.target as HTMLIonInputElement).value;
    setCriteria(term);
    emitter.next(term);
  };

  useEffect(() => {
    const term$        = emitter.asObservable();
    const allContacts$ = service.getContacts().pipe(takeUntil(term$));
    const searchTerm$  = service.autoSearch(term$);
    const watch = merge(allContacts$, searchTerm$).subscribe(setPeople);

    return () => watch.unsubscribe();
  }, [service, setPeople]);
```

Lab Exercise

Manual subscribe()

```
export const ContactsList: React.FC = () => {
  const [people, setPeople] = useState<Contact[]>([]);
  const [criteria, setCriteria] = useState<string>('');
  const [service] = useState<ContactsService>(injector.get(ContactsService));
  const [emitter] = useState<Subject<string>>(new Subject<string>());

  const doSearch = (e: Event) => {
    const term = (e.target as HTMLIonInputElement).value;
    setCriteria(term);
    emitter.next(term);
  };

  useEffect(() => {
    const term$        = emitter.asObservable();
    const allContacts$ = service.getContacts().pipe(takeUntil(term$));
    const searchTerm$  = service.autoSearch(term$);
    const watch = merge(allContacts$, searchTerm$).subscribe(setPeople);

    return () => watch.unsubscribe();
  }, [service, setPeople]);

  return ( ... );
}
```

Speaker notes

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

Traditional **Unsubscribe**()

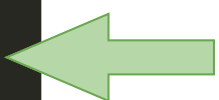
```
export const TicketsList extends React.Component {
  constructor() {
    this.setState({
      watch  : null,
      tickets: []
    });
  }

  componentDidMount() {
    const term$ = emitter.asObservable();
    const searchTerm$ = service.autoSearch(term$);
    const allContacts$ = service.getContacts().pipe(takeUntil(term$));
    const watch = merge(allContacts$, searchTerm$).subscribe(
      tickets => this.setState({ tickets })
    );

    this.setState({watch})
  };

  componentWillUnmount() {
    watch.unsubscribe();
  }

  return (
    <IonList>
      {tickets.map((ticket, idx) => {
        return <TicketListItem key={idx} ticket={ticket} />;
      })}
    </IonList>
  );
};
```

This is a typical solution seen in the community.

This is not good... avoid doing this!

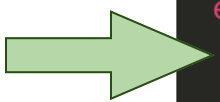
Coding with **Observables**

```
export class ContactsList extends React.Component {
    constructor({emitter, service}) {
      this.state = {
        watch: null,
        contacts: [],
        emitter,
        service
      };
    }

  componentDidMount() {
    const term$        = this.emitter.asObservable();
    const searchTerm$  = this.service.autoSearch(term$);
    const allContacts$ = this.service.getContacts().pipe( takeUntil(term$) );

    const watch = merge(allContacts$, searchTerm$).subscribe(
      contacts => this.setState({contacts})
    );

    this.setState({watch});
  }

  componentWillUnmount() {
    this.state.watch.unsubscribe();
  }
}
```



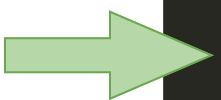
Best Practice: Almost **never** manually subscribe to Observables.

Using the **useObservable** custom hook



```
export type InputEmitter = Subject<string>;

export const ContactsList: React.FC = () => {
  const [contacts, setStream] = useObservable<Contact[]>(null, []);
  const [service] = useState<ContactsService>(injector.get(ContactsService));
  const [emitter] = useState<InputEmitter>(() => new Subject<string>());


  useEffect(() => {
    const term$        = emitter.asObservable();
    const searchTerm$  = service.autoSearch(term$);
    const allContacts$ = service.getContacts().pipe(takeUntil(term$));
    const merged$      = merge(allContacts$, searchTerm$);

    setStream(merged$);

  }), [emitter, service, setContacts$]);

  return (
    // ....
  );
}
```

Why is **useObservable** so important?



```tsx
export const ContactsList: React.FC = () => {
  const [service] = useState(() => new ContactsService());
  const [contacts, setContacts$] = useObservable<Contact[]>(null, [])

  useEffect(() => {
    setContacts$(service.getTickets());
  }, [service, setContacts$]);

  return (
    <IonList>
      {contacts.map((contact, idx) => {
        return <ContactListItem key={idx} contact={contact} />;
      })}
    </IonList>
  );
};
```

- **Subscribes** to specified observable
- **Extracts** the value to the specified variable
- **Triggers** view rerenders
- Auto-**unsubscribe**() during unmount
- Auto-unsubscribe + Subscribe to new Observable

useObservable React Hook

```
export function useObservable<T>(observable$: Observable<T>, initialValue?: T): [T | undefined, ResetStreamSource<T>] {
  const [source$, setObservable] = useState<Observable<T>>(observable$);
  const [value, setValue] = useState<T | undefined>(initialValue);
  const reportError = (err: any) => console.error(`useObservable() error: ${JSON.stringify(err)}`);

  useEffect(() => {
    if (source$) {
      const s = source$.subscribe(setValue, reportError);
      return () => {
        s.unsubscribe();
      };
    }
  }, [source$]);

  return [value, setObservable];
}
```

@mindspace-io/react

axios-observable

- 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

axios-observable

- 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()

Speaker notes

Okay, so let’s talk about using Axios for robust REST services
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

RxJS Lab 5: useObservable() Hook

```
export type InputEmitter = Subject<string>;

export const ContactsList: React.FC = () => {
  const [contacts, setStream] = useObservable<Contact[]>(null, []);
  const [service] = useState<ContactsService>(injector.get(ContactsService));
  const [emitter] = useState<InputEmitter>(() => new Subject<string>());


  useEffect(() => {
    const term$        = emitter.asObservable();
    const searchTerm$  = service.autoSearch(term$);
    const allContacts$ = service.getContacts().pipe(takeUntil(term$));
    const merged$      = merge(allContacts$, searchTerm$);

    setStream(merged$);

  }), [emitter, service, setContacts$]);

  return (
    // ....
  );
}
```

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


```
export const injector: DependencyInjector = makeInjector([
  {
    provide: ContactsService,
    useValue: {
      getContacts: () => of([]) // empty list
    }
  }
]);
```

Here we are building a mock REST service!

Creating your Own Observable

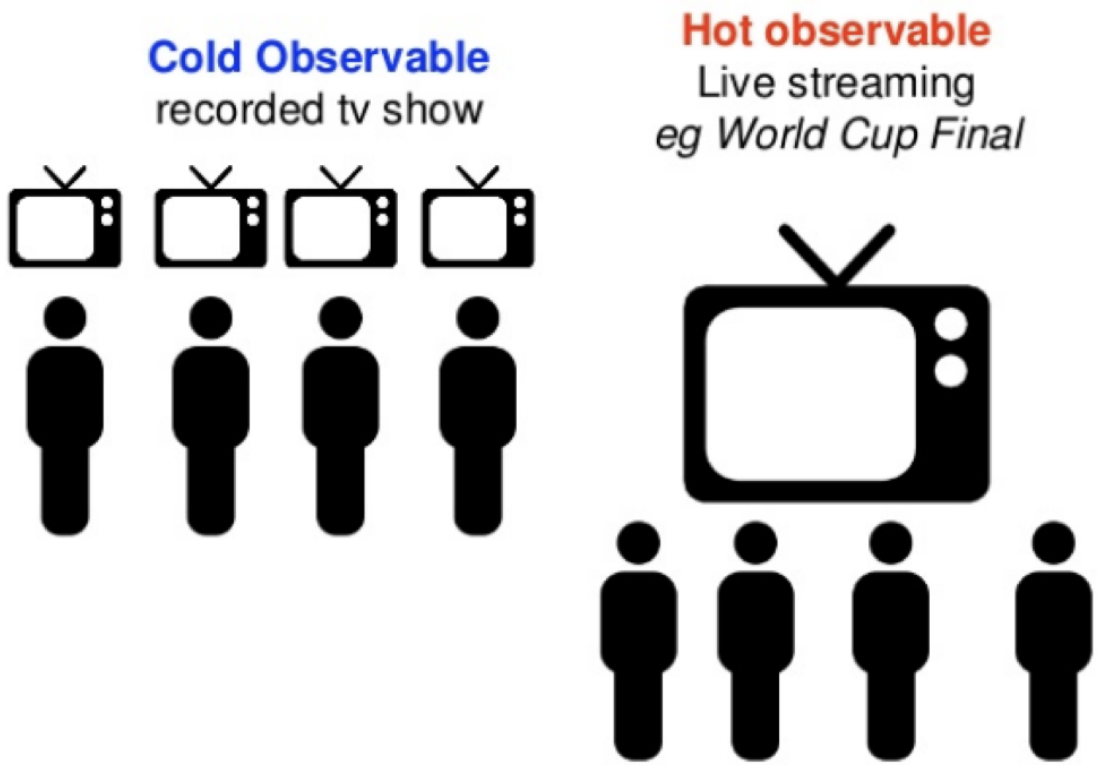
4

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

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

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

  observer.complete();

});
```

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, Observables are COLD



Cold Observer: each subscriber is called with new activity.

Things to consider:

- **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


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

const custom$ = new Observable( subscriber => {
  const status = "ALL GOOD";

  const intervalID = setInterval(() => {
    try {
        subscriber.next( status);
    } catch(e) {
      subscriber.error( 'oops' );

  }, 1000);

  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

4

Unicast

Observable only send notifications to a single Observer.

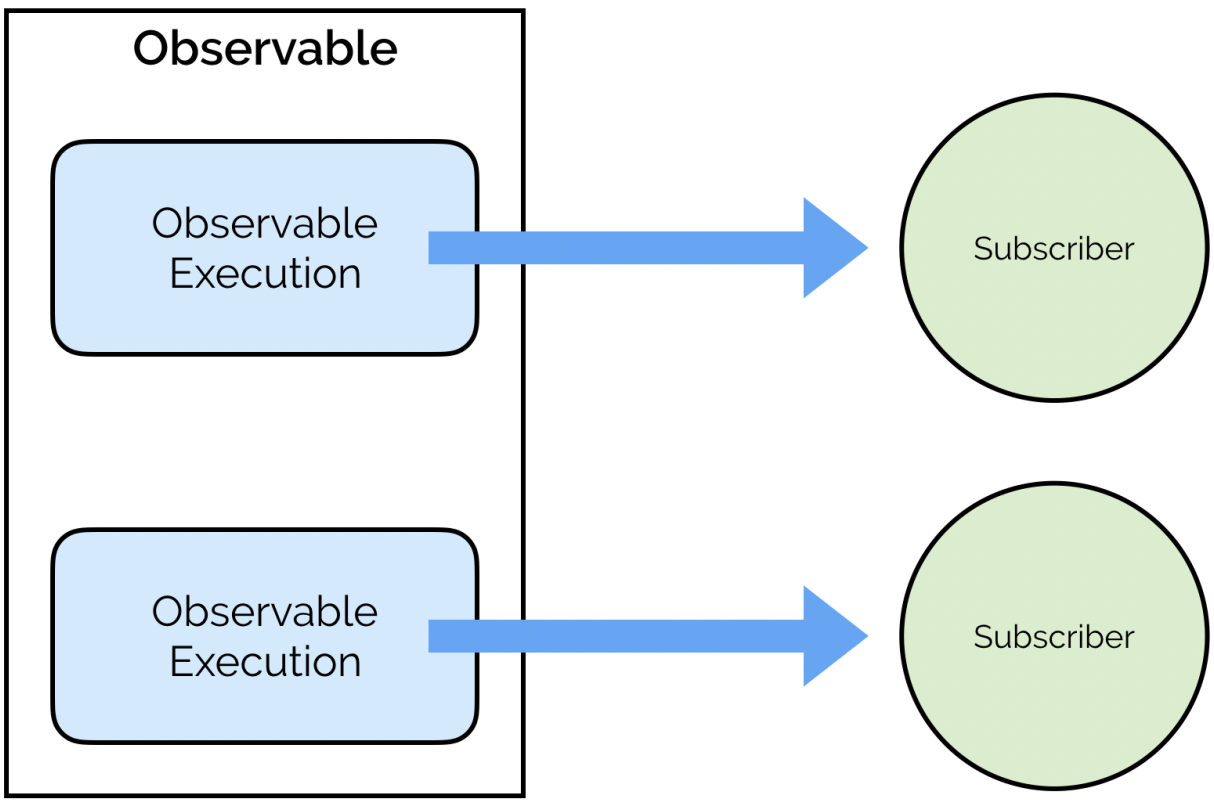
e.g. HttpClient

Multicast

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

e.g. **EventEmitter**

Unicast



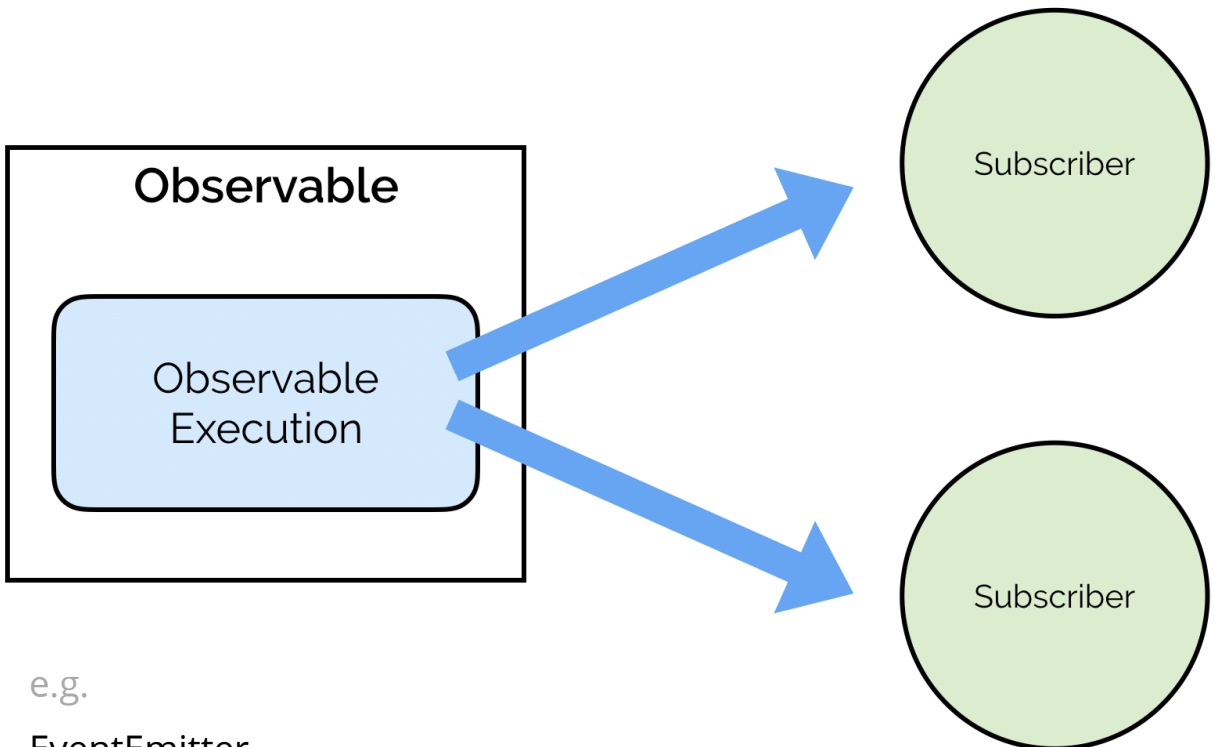
e.g. HttpClient

Speaker notes

Each subscribe runs a new observable execution
ContactsService (with promised-based fetch) methods do this.

Now we see why an Service methods could be called multiple times...

Multicast



e.g.

EventEmitter

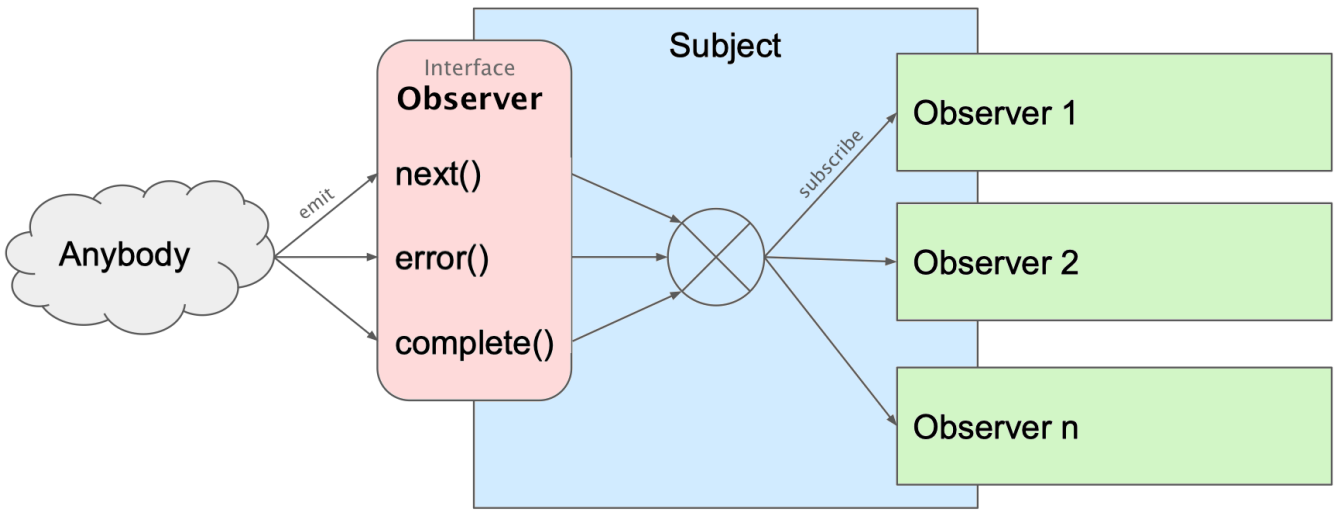
FormControl.valueChanges

Speaker notes

multicast share an Observable Execution amongst subscribers

Like BehaviorSubject they retain a registry of listeners

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**...

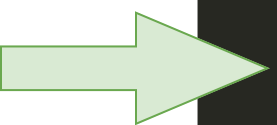
uses **last-emitted value** for new subscribers...

Speaker notes

Subject is a hot observable.
BehaviorSubject is a warm observable ;-)

Multicast: Subject

4

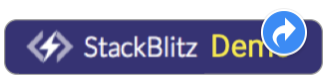


```
export const ContactsList: React.FC = () => {
  const [contacts, setContacts] = useState<Contact[]>([]);
  const [service] = useState(() => new ContactsService());

  const [emitter] = useState(() => new Subject<string>());

  useEffect(() => {
    const term$ = emitter.asObservable();
    const watch = term$.subscribe(value => {
        const request$ = service.searchBy(value);
        request$.subscribe( setContacts );
    });

    return () => watch.unsubscribe();
  }, [emitter, service, setContacts]);

  return (   ...  );
};
```

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);
```



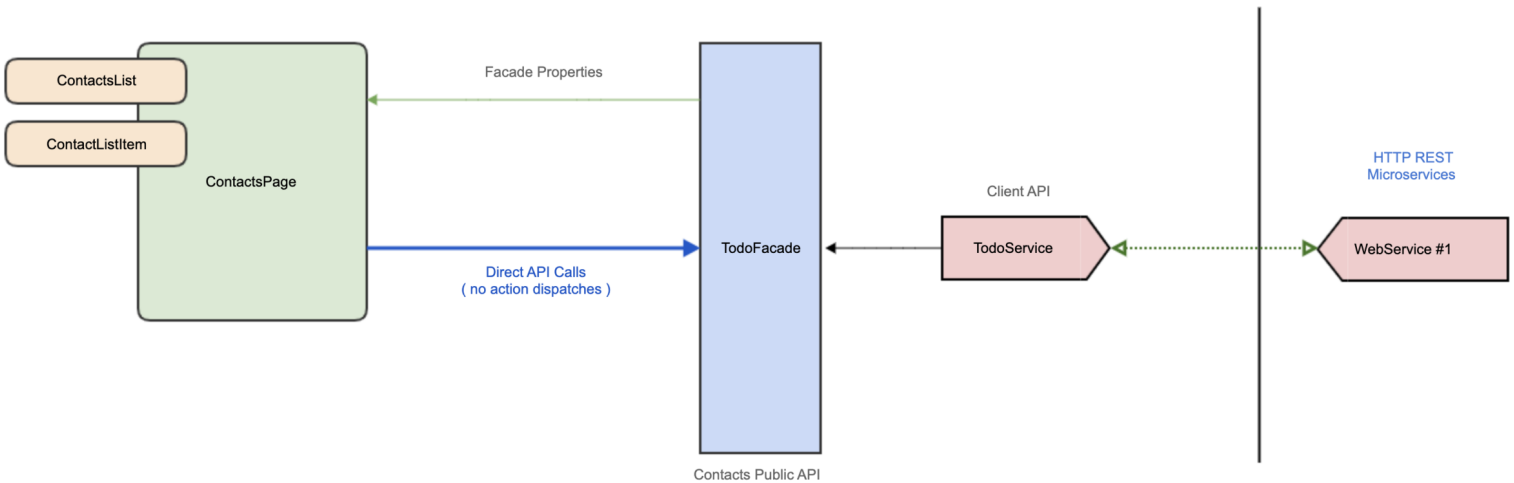
Speaker notes

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

Facades (Push-based)

- Services to load & manipulate data **for views** are Facades
- Facades **hide complexity** and internal activity
- Facades provide simple, clear, terse API(s)
- Be consistent in Facades with Observable API(s)
- **Never build** APIs with **both** Observables + raw-data

Facades

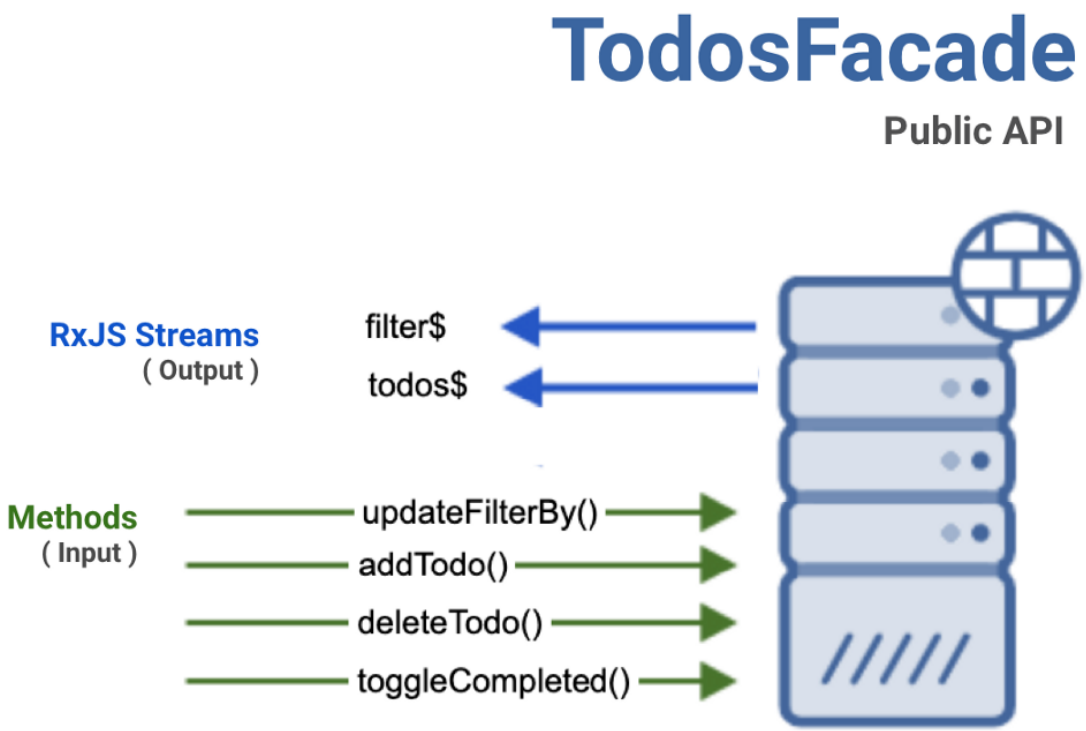


- Services to load & manipulate data **for views** are Facades
- Facades **hide complexity** and internal activity
- Facades provide simple, clear, terse API(s)

Facades

- Pull-based (Promises or 'raw' data')
- Push-based (RxJS, Observables)
- Be consistent in Facades with Observable API(s)
- **Never build** Facade API with **both** Observables + raw-data

Facades


TodosFacade
Public API
RxJS Streams
(Output)
filter$
todos$
Methods
(Input)
updateFilterBy()
addTodo()
deleteTodo()
toggleCompleted()

Facade with RxJS

```
export type InputEmitter = Subject<string>;

export class ContactsFacade {
  private emitter: InputEmitter;

  readonly contacts$: Observable<Contact[]>;
  readonly criteria$: Observable<string>;

  constructor(private service: ContactsService) {
    const emitter = new Subject<string>();
    const term$ = emitter.asObservable();
    const searchByCriteria$ = service.autoSearch(term$);
    const allContacts$ = service.getContacts().pipe(takeUntil(term$));

    this.emitter = emitter;
    this.criteria$ = term$;
    this.contacts$ = merge(searchByCriteria$, allContacts$);
  }

  searchFor(partial: string): Observable<Contact[]> {
    //...
  }

  selectById(id: string): Observable<Contact | undefined> {
    //...
  }
}
```

- API is simple and designed for views
- Read-only streams emit data
- Views connect to streams and
- Views react to stream data
- Separation of event delegation from data delivery
- Methods trigger internal activity
  - Internal details are hidden
  - Method calls and stream emissions are **asynchronous**
- Methods **never** return raw data

Thomas Burleson
Jan 14 · 8 min read ★

React Best-Practices: Facades

State Management

182

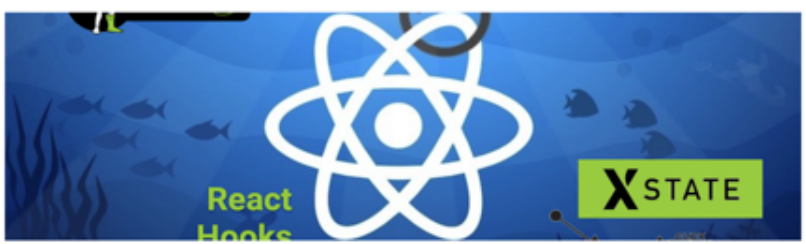
Thomas Burleson
Jul 27, 2019 · 5 min read

React Hooks + RxJS

Use React Hooks with RxJS Facades to create push-based architectures and radically...

685 4 responses

Thomas Burleson
Jan 4 · 5 min read



React Custom Hooks & Animations

State machines ensure components are constrained to specific states...

24

Thomas Burleson
Jan 19 · 5 min read

The Power of Tuples

Learn about Typescript Tuples for React and Angular applications.

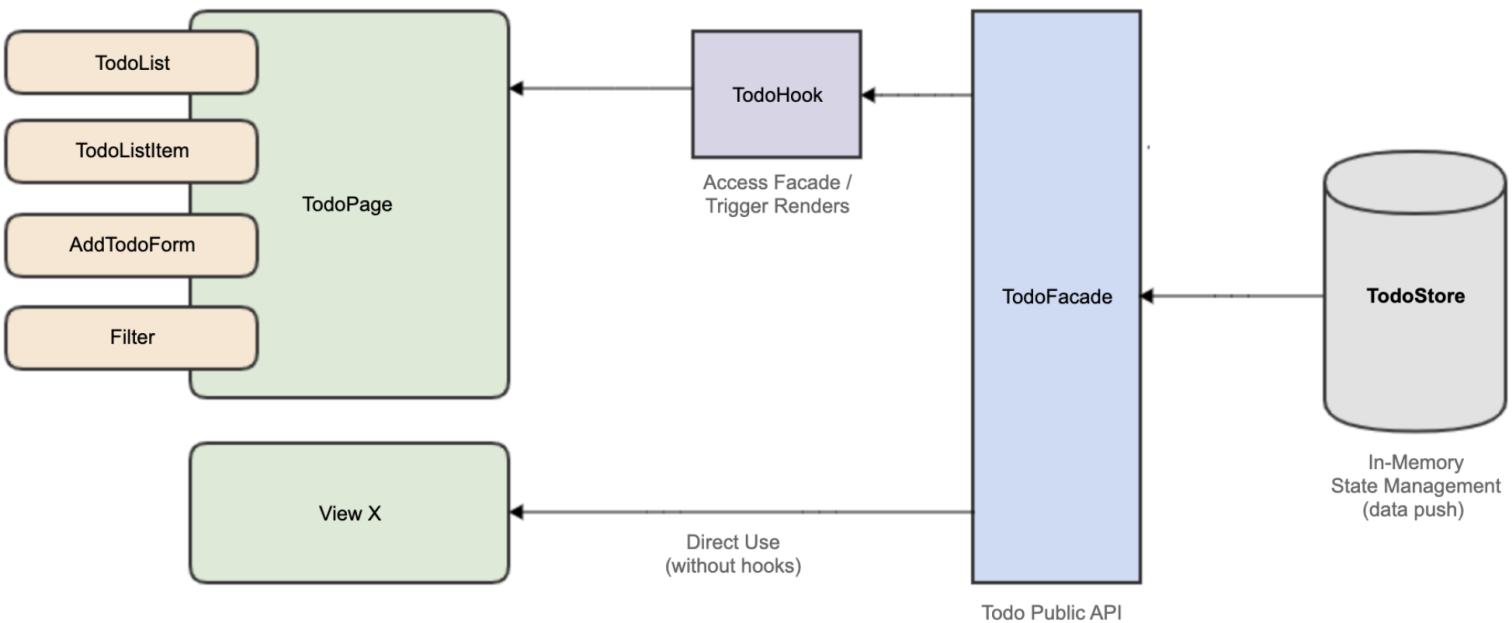
32

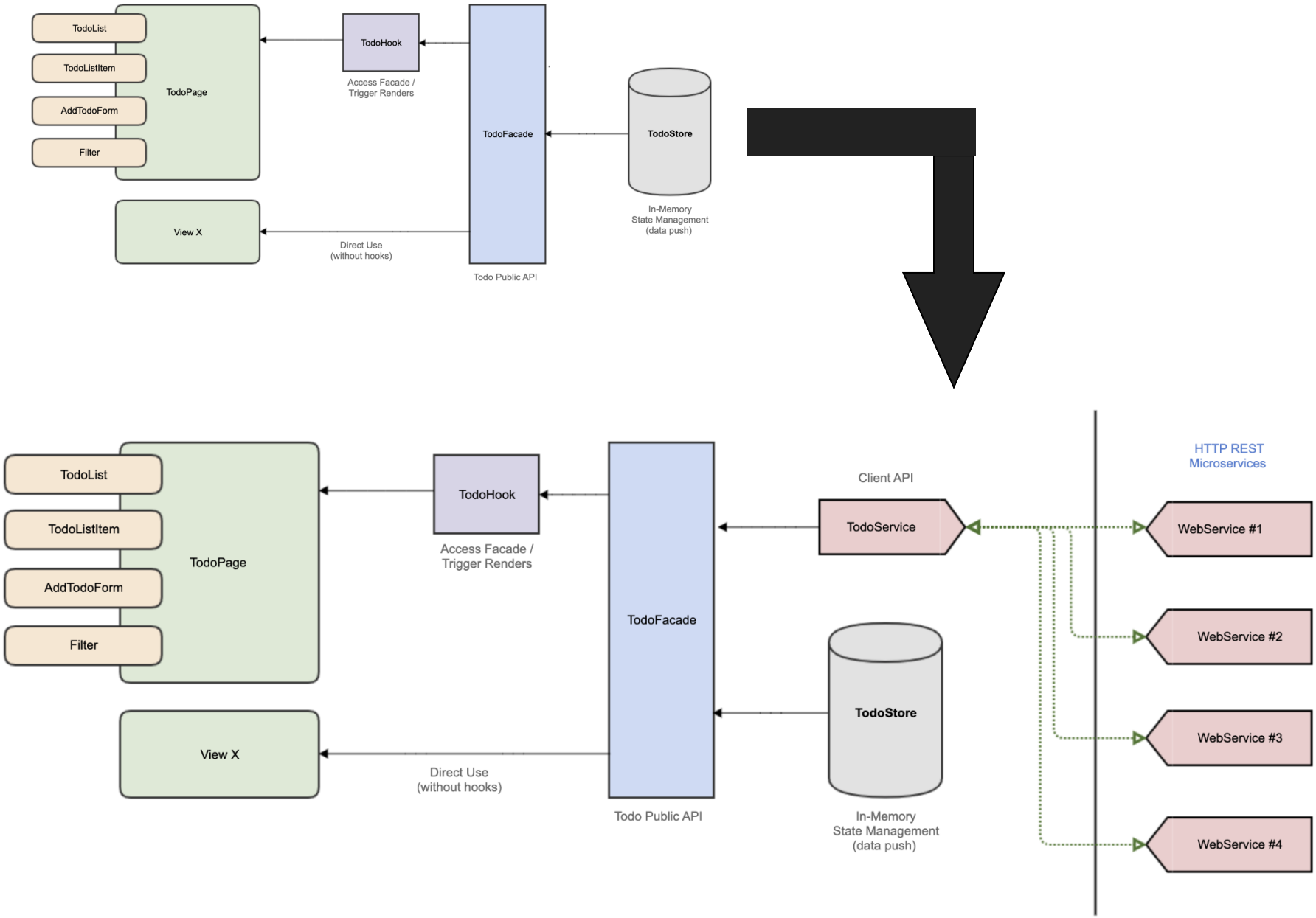
RxJS Lab 6: Reactive Facades

```
export type InputEmitter = Subject<string>;

export class ContactsFacade {
  private emitter: InputEmitter;

  readonly contacts$: Observable<Contact[]>;
  readonly criteria$: Observable<string>;

  constructor(private service: ContactsService) {
    const emitter = new Subject<string>();
    const term$ = emitter.asObservable();
    const searchByCriteria$ = service.autoSearch(term$);
    const allContacts$ = service.getContacts().pipe(takeUntil(term$));

    this.emitter = emitter;
    this.criteria$ = term$;
    this.contacts$ = merge(searchByCriteria$, allContacts$);
  }

  searchFor(partial: string): Observable<Contact[]> {
    //...
  }

  selectById(id: string): Observable<Contact | undefined> {
    //...
  }
}
```

View + Hooks + Facades

aka *View-Layer* Services




TodoList
TodoListItem
AddTodoForm
Filter
TodoPage
TodoHook
Access Facade /
Trigger Renders
TodoFacade
TodoStore
In-Memory
State Management
(data push)
View X
Direct Use
(without hooks)
Todo Public API
Client API
TodoService
HTTP REST
Microservices
WebService #1
WebService #2
WebService #3
WebService #4

Custom Hooks

- **Move** all business logic + state management outside view components
- **Used** with Facades
- **Used** with `useObservable()` + RxJS

Custom Hooks

Important to declare Tuple types:

```
/**
 * Define Tuples for Reactive Hooks
 */
export type ContactsTuple = [string, Contact[], ContactsFacade];
export type ContactDetailsTuple = [Contact, { goBack: () => void }];


/**
 * Export custom hook for contacts-list.tsx
 */
export function useContacts(): ContactsTuple {

  return [...];
}
```

...used in the views!

Using **tuples** in views:

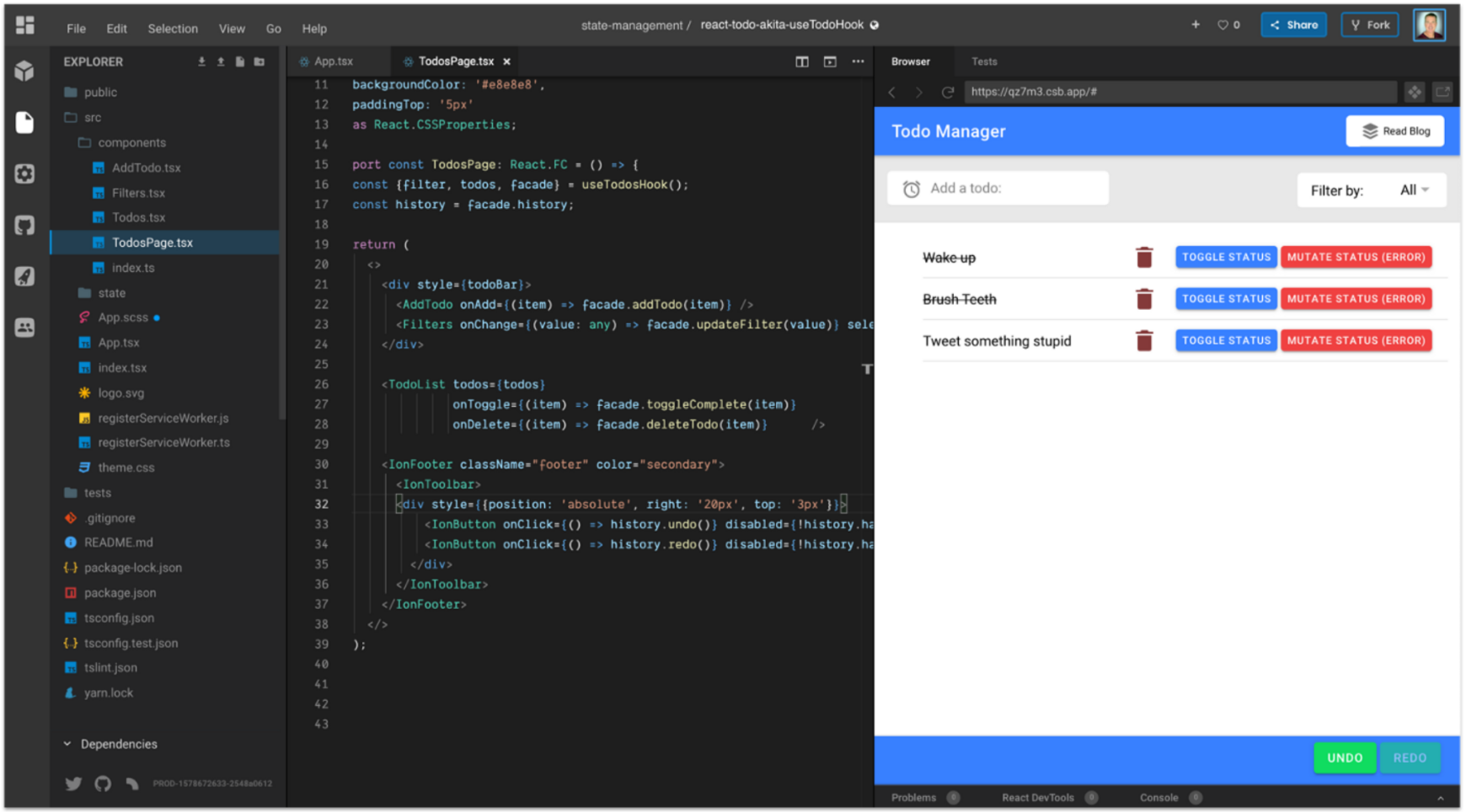
```tsx
import { useContacts, ContactsTuple } from '@workshop/contacts/data-access';

export const ContactsList: React.FC = () => {
  const [criteria, people, facade]: ContactsTuple = useContacts();

  return ( ... );
}
```

contacts-list.tsx

IDE(s) support type hints...

```
19
20    export const Contac  const people: Contact[]
21      const [criteria, people, facade] = useContacts();
22
```

Live Demo



https://bit.ly/**34vfl2E**

RxJS Lab 7: Reactive Hooks

```tsx
export type ContactsTuple = [string, Contact[], ContactsFacade];
export type ContactDetailsTuple = [Contact, { goBack: () => void }];

/**
 * For use in contacts-list.tsx
 */
export function useContacts(): ContactsTuple {
  const [facade] = useState<ContactsFacade>(() => injector.get(ContactsFacade));
  const [criteria] = useObservable(facade.criteria$, '');
  const [contacts] = useObservable(facade.contacts$, []);

  return [criteria, contacts, facade];
}

/**
 * For use in contact-detail.tsx
 */
export function useContactDetails(): ContactDetailsTuple {
  const { id } = useParams();
  const [facade] = useState<ContactsFacade>(() => injector.get(ContactsFacade));
  const [contact, setContact$] = useObservable<Contact>(null, {} as Contact);
  const history = useHistory();

  useEffect(() => {
    setContact$(facade.selectById(id));
  }, [id, facade]);

  return [contact, history];
}
```