Deep Dive into RxJS

David Benson

smartdata

Who Am I?

David Benson

david.benson@smartdata.net

I work at Smart Data, a fun and growing software development consultancy in Dayton and Cincinnati Ohio

(We're hiring!)

Passionate about tech

smartdata

What is RxJS?

How does it work?

What is it good for?

smartdata

What is RxJS?

RxJS is a implementation of Reactive Extensions (ReactiveX) for Javascript and TypeScript

ReactiveX was developed by Erik Meijer at Microsoft

Open sourced in 2012

Used by GitHub, Netflix, SoundCloud, AirBnb and others

Reactive Extensions is Cross Platform (available for Java, C#, C++, JavaScript, Python, and more)

http://reactivex.io/

The Observer pattern done right

ReactiveX is a combination of the best ideas from
the Observer pattern, the Iterator pattern, and functional programming

smartdata

The Observer Pattern

The **Observer Pattern** is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

smartdata

The Iterator Pattern

The **Iterator Pattern** provides a way to access the elements of an object container sequentially without exposing the underlying representation.

smartdata

Functional Programming

Functional programming is a declarative style of building software by using pure functions, avoiding shared state, mutable data, and side-effects.

smartdata

What is ReactiveX?

Simplified asynchronous functional data streams

What is ReactiveX?

Observable objects

That can be subscribed to by other methods

And then return an asynchronous stream of updates

Which can be modified through functions

Isn't that like promises?


RxJS
RxJS

RxJs

smartdata

How are Rx Observables different from Promises?

1. An observable handles a stream of async events, instead of a single event.

```
let a = new Observable(observer => {
    setTimeout(() => observer.next(1), 1000);
    setTimeout(() => observer.next(2), 2000);
    setTimeout(() => observer.next(3), 3000);
    setTimeout(() => observer.complete(), 4000);
});
a.subscribe(result => console.log(result));
```

```
1
2
3
```

smartdata

How are Rx Observables different from Promises?

2. An observable is Lazy

```
let a = new Observable(observer => {
    setTimeout(() => observer.next(1), 1000);
    setTimeout(() => observer.next(2), 2000);
    setTimeout(() => observer.complete(), 3000);
});
setTimeout(() => {
     console.log(3);
    a.subscribe(result => console.log(result));
}, 3000)
```

```
3 The observable hasn't started yet
1
2
```

smartdata

Charlie Kaufman's observables won't ever execute.

How are Rx Observables different from Promises?

3. An observable is Cancellable

takeUntil(observable), takeWhile(predicate), take(n), first(), first(predicate)

```
let a = new Observable(observer => {
    setTimeout(() => observer.next(1), 1000);
    setTimeout(() => observer.next(2), 2000);
    setTimeout(() => observer.next(3), 3000);
    setTimeout(() => observer.complete(), 4000);
});
let b = new Observable(observer => {
    setTimeout(() => observer.next(), 2500);
    setTimeout(() => observer.complete(), 4000);
});
a.pipe(takeUntil(b))
 .subscribe(
    result => console.log(result)
  );
```

```
1
2
```

smartdata

How are Rx Observables different from Promises?

4. An observable is Retryable

```
let a = new Observable(observer => {
    observer.next(1);
    throw "Error!";
    setTimeout(() => observer.complete(), 4000);
});
a.pipe(retry(3))
 .subscribe(
    result => console.log(result),
    err    => console.log("Error")
 );
```

```
1
1 retry 1
1 retry 2
1 retry 3
Error
```

smartdata

But... Rx Observables and promises are compatible!

RxJs operators that accept observables can accept promises as well.

So no need to replace everything with an observable

Convert Promises to Observables with

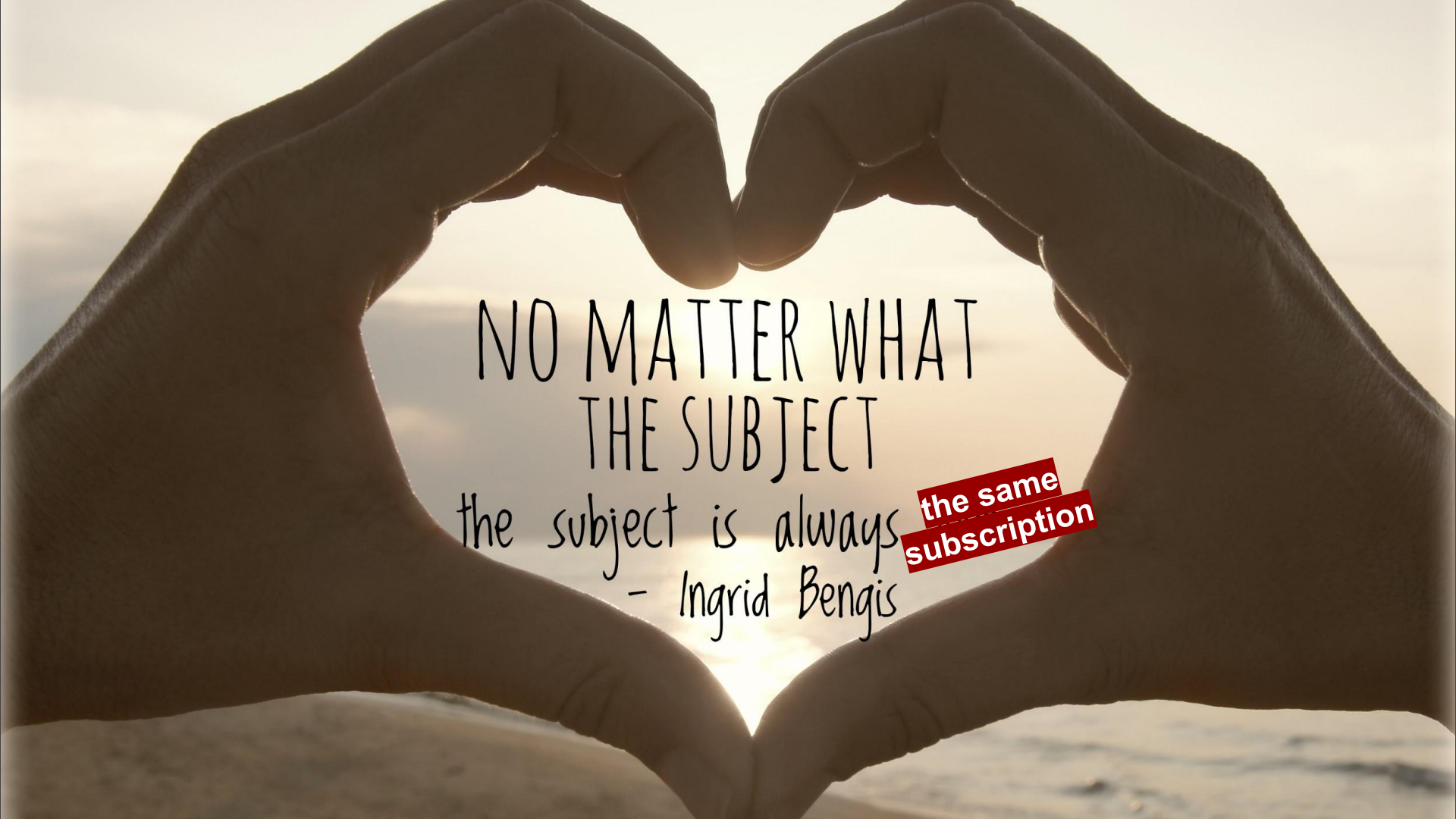
```
Observable.defer(x => promise())
```

smartdata

Observables and Subjects

Observables are just a base class for RxJs, there are many derived types

Subjects are the main derived type from Observables


NO MATTER WHAT
THE SUBJECT
the subject is always
- Ingrid Bengis
the same
subscription

Observables and Subjects

Observable

Observables will only execute upon subscribe, and will re-execute every time they are subscribed.

```
let a = new Observable(observer => {
    console.log(1);
    observer.next(2);
});
a.subscribe(result => console.log(result));
a.subscribe(result => console.log(result));
```

```
1
2 first result
1
2 second result
```

Observables and Subjects

Subject

Subjects are observables, that are also observers. They will send updates to all subscriptions, and allow updating from external sources.

```
let a = new Subject();
let b = a.pipe(tap(() => console.log('Side Effect')));
b.subscribe(result => console.log(result));
b.subscribe(result => console.log(result));
console.log(1);
a.next(2);
```

```
1
Side Effect
2 first result
2 second result
```

Subjects

BehaviorSubject

A subject that stores the latest value, and immediately sends it to new subscribers.

ReplaySubject

A subject that stores the latest x values, and immediately sends those values to new subscribers.

AsyncSubject

A subject that waits until it is complete, then sends the final value to any subscribers (even latecomers).

ConnectableObservable

A wrapper that makes an observable behave like a subject, but with a .connect() function to start it.

Multicasting

To convert an Observable to a Subject, you can use the multicast operator:

```
let a = new Observable(observer => {
    console.log(1);
    observer.next(2);
}).pipe(
    multicast(() => new Subject())
);
a.subscribe(result => console.log(result));
a.subscribe(result => console.log(result));
a.connect(); // fires the initial observable
```

```
1
2 first result
2 second result
```

Multicast creates a new ConnectableObservable, then subscribes to the original observable, passing the events through.

RxJs has shortcut methods to create specific multicast patterns:

publish: Multicast to a new Subject

share: Multicast to a new Subject, but if the Subject completes or errors, recreate the Subject for new subscribers.

Our powers combined

RxJS operators can be super powerful.

Using multiple observables together is even better!

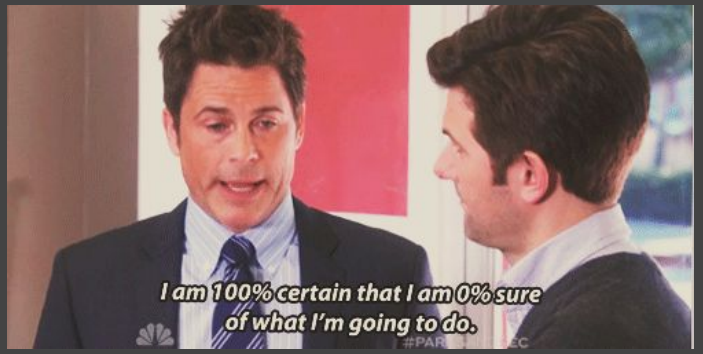
RxJs has operators that let you either combine observables into an array, combine them into a single stream, or transform an observable into another one.

smartdata

SO MANY TO CHOOSE

RxJS has a lot of operators

It can be hard to figure out what you need



So let's try them all out with Marble Diagrams!

smartdata

Joining Observables Into An Array

combineLatest ★

Begins once all observables have fired once
Fires whenever any observable sends an event

forkJoin

Return the last value from all observables once they have all completed.

zip

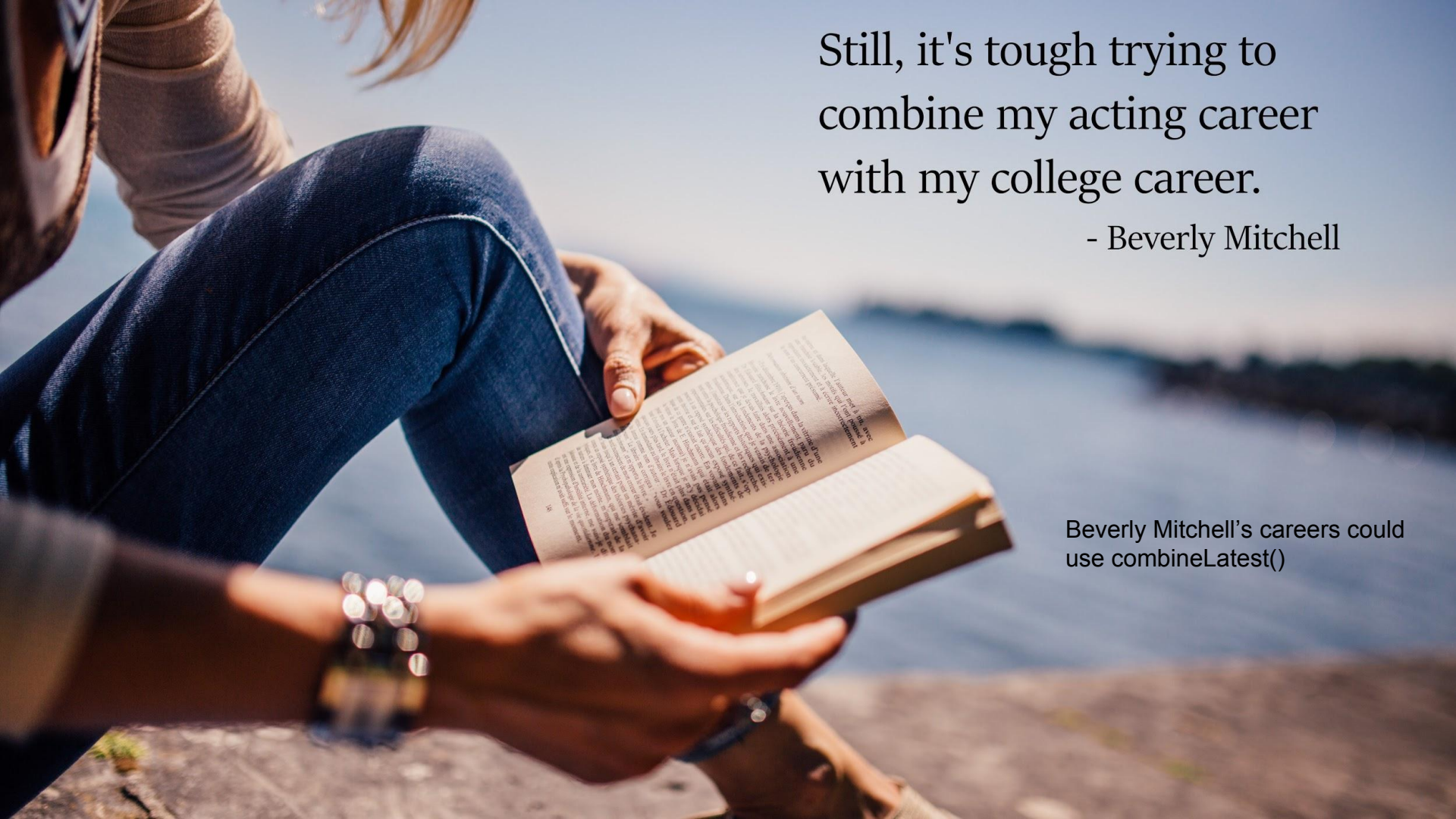
Begins once all observables have fired once
Fires 1:1 when all observables have sent the next event

withLatestFrom (piped)

Begins when the target has been fired after the source has been given a value.
Fires whenever any observable sends an event

https://rxviz.com/v/Ro3V5M98

smartdata



Beverly Mitchell’s careers could use combineLatest()

Joining Observables Into A Single Stream

merge 

Passes through values from multiple observables into a single observable
Passes through all values from any observable

race

Watches all of the given observables until one emits a value, then passes through that observable and discards the others.

concat (piped)

Calls a series of observables in order, waiting for one to finish before calling the second

startWith (piped)

give a starting value to an observable

https://rxviz.com/v/qJwKvGpJ

smartdata

Mapping Observables To New Observables

smartdata

Mapping Observables To New Observables

mergeMap (piped)

Subscribe to every new observable that we return from the map.

concatMap (piped)

Queues up observables, and then subscribes to a new observable when the previous observable completes.

switchMap (piped) 

Unsubscribe from the previous observable when the new one begins.

exhaustMap (piped)

Ignores new Observable events until the current Observable completes. Opposite of SwitchMap

https://rxviz.com/v/dJB607gJ

smartdata

Custom Pipe Operators

smartdata

Custom Pipe operators

RxJS has many operators already defined, that should handle most common use cases - (map, filter, debounce, etc)

If you need something special, you can create your own

Like the observable itself, pipe operations are not executed until the observable is subscribed.

Custom Pipe operations

wrapper => function => new observable

```
const splitString = (splitOn = ',') =>
  (source) =>
    new Observable(observer => {
      return source.subscribe({
        next(x)    { observer.next(x.split(splitOn)); },
        error(err) { observer.error(err); },
        complete() { observer.complete(); }
      });
    });
```

Create a wrapper
*That returns a **Function***
*Which returns a new **Observable***
*Based on the input **Observable***

Make sure to pass errors
And to clean up

```
const splitString =
  (splitOn = ',') => map(x => x.split(splitOn));
```

Just wrap map

Custom Pipe operators - in the real world

```
const thenCombineLatest = func => source => {
    let p = source.pipe(publish(),refCount())
    return new Observable(observer => p.subscribe({
        next(x) {
            let f = func.call(this, x);
            f = Array.isArray(f) ? f : [f];
            x = Array.isArray(x) ? x : [x];
            setTimeout(() => combineLatest.call(this, f).pipe(takeUntil(p))
                .subscribe(r => observer.next([...x, ...r])));
        },
        error: observer.error,
        complete: observer.complete
    }));
}
```

```
getUsers('David').pipe(
    thenCombineLatest(user => getKids(user)),
    thenCombineLatest(([user,kids]) => getHumorStyle(kids.length))
).subscribe(([user,kids,humorStyle]) => {})
```

https://rxviz.com/v/qJyAE4EJ

smartdata

Using RxJs with Frameworks

smartdata

Framework Support

Angular: use the async pipe:

```
<div *ngIf="product | async; let product">
     <h1>{{product.name}}</h1>
     <p>{{product.description}}</p>
</div>
```

React: Use Subjects instead of Actions and Stores, while keeping a unidirectional data flow. http://www.thepursuit.io/2018/02/why-to-use-rx-with-react/

Vue: Use Vue-rx to wire up subscriptions.
https://codeburst.io/combining-vue-typescript-and-rxjs-with-vue-rx-a084d60b6eac


I do not like to repeat successes,
I like to go on to other things.
-Walt Disney
Preventing repeats

Preventing repeats

- Use a multicast operator (publish, share, or shareReplay) to prevent multiple executions
- Just subscribe once (use pipes to modify subscriptions and handle nested subscriptions)
- Filter out irrelevant values

Preventing Memory Leaks

smartdata

Watch out for memory leaks

Observables and Subjects are cleaned up when they are completed.

If a subject is never completed, it will not be released, leading to memory leaks.

- Make sure to unsubscribe from open Subjects or Observables in teardown methods - (ngOnDestroy, componentWillUnmount)
- Use a base class and helper function to manage Subjects or Observables and teardown
- Use takeUntil to complete all open observables with a single line in the teardown method

TakeUntil demo

https://codesandbox.io/s/9j2xkrzpzp

Questions?

smartdata

Resources

https://rxviz.com

http://rxmarbles.com

https://medium.com/@benlesh

smartdata