Taming space and time with RxJS Observables

MidwestJS 2017

Jeff Barczewski

- @jeffbski
- jeff@codewinds.com
- https://codewinds.com/mwjs2017

Jeff Barczewski

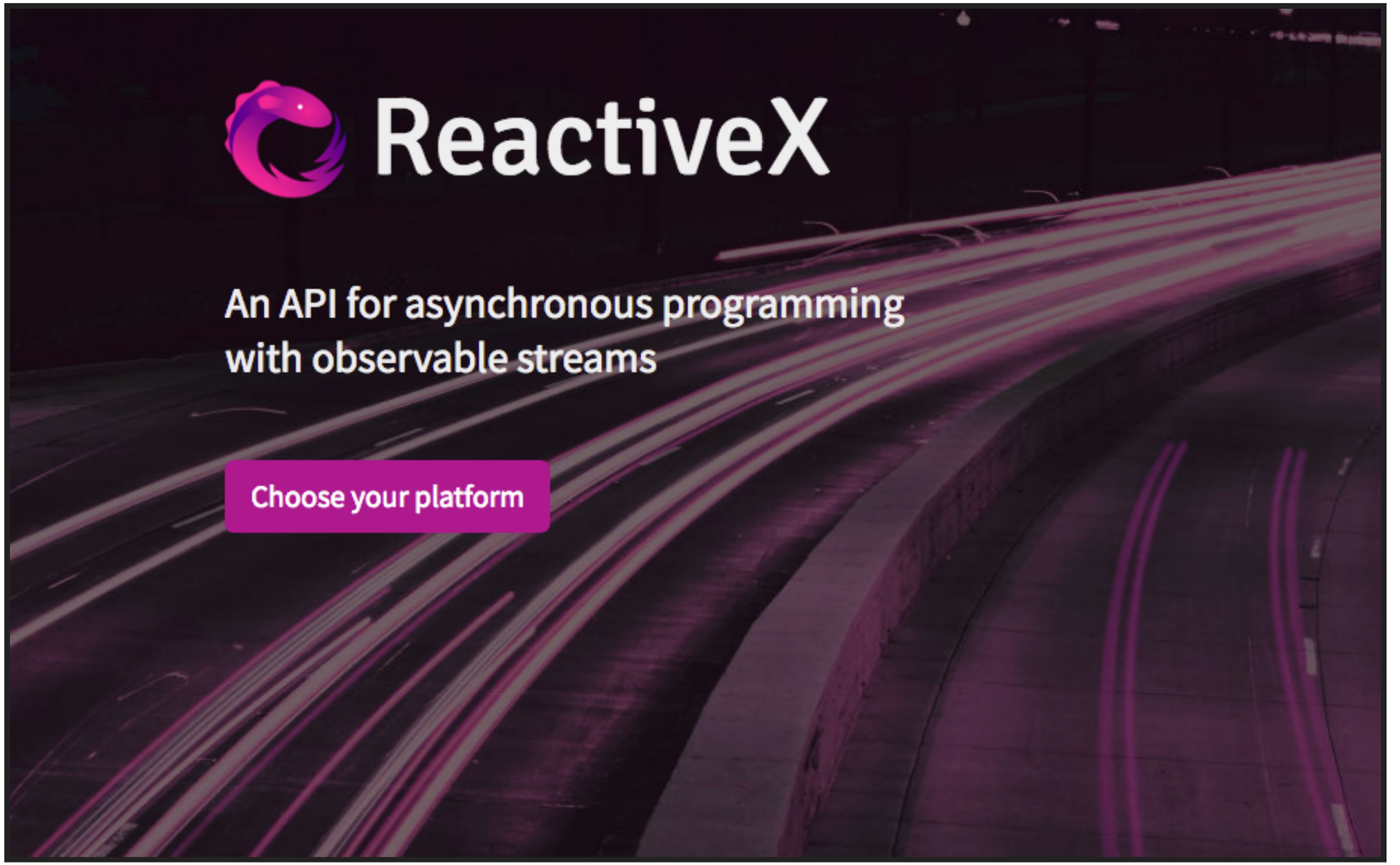
- Married, Father, Catholic
- 27 yrs (be nice to the old guy :-)
- JS (since 95, exclusive last 5 years)
- Open Source: redux-logic, pkglink, …
- Work: OCI, MasterCard ApplePay, Elsevier, RGA
- Founded CodeWinds, live/online training (React, Redux, Immutable, RxJS) – I love teaching, contact me

CodeWinds Training

- Live training (in-person or webinar)
- Self-paced video training classes (codewinds.com)
- Need training for your team on any of these?
  - React
  - Redux
  - RxJS
  - JavaScript
  - Node.js
  - Functional approaches
- I'd love to work with you and your team
- I appreciate any help in spreading the word.

Foundational Questions

- What is RxJS?
- Why should I care?


ReactiveX
An API for asynchronous programming
with observable streams
Choose your platform

Back in 1995

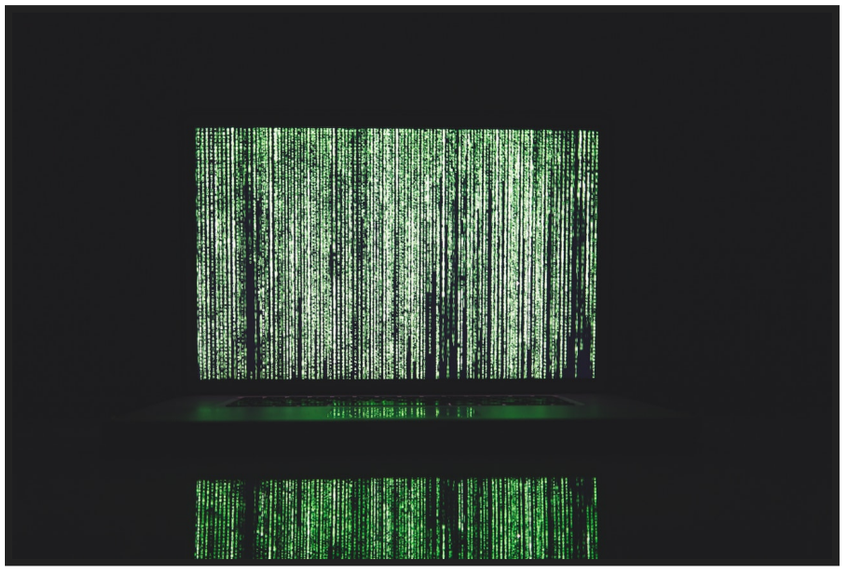
1. **Browser** makes **single request** to server
2. **Server** responds with full page of **HTML**
3. **Browser** renders the **HTML**
4. User clicks link, repeat starting with 1

Today (w/o HTTP/2)

1. **Browser** makes **initial** request to server
2. **Server** responds with **partial** page
3. **Browser** fetches lots of **js** and **data**
4. **Server(s)** respond with the **js** and **data**
5. **Browser** runs **js** and might fetch **more data**
6. **Browser** renders incomplete page
7. **Server(s)** responds with additional data
8. **Browser** upgrades connection to a **web socket**
9. **Server** pushes data over web socket as it is available
10. **Browser** updates page as **new data** comes in

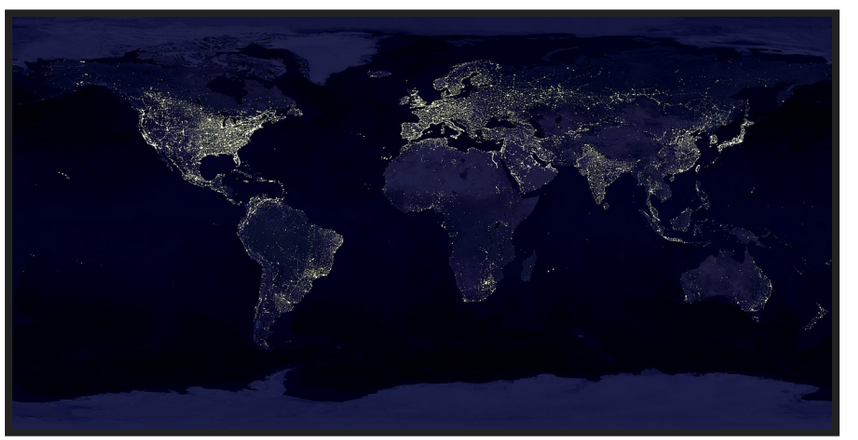
Today with HTTP/2

1. **Browser** makes **initial request** to server
2. **Server** responds with partial page and **predicted js/data**
3. **Browser** makes **N concurrent** requests using same connection to get additional data
4. **Server** responds to the concurrent requests over the same connection
5. **Browser** upgrades connection to a **web socket**
6. **Server** pushes data as it is available
7. **Browser** updates page as **new data** comes in



Today's Challenges

- Data needs to be pushed in real time
- Apps and pages hold state to improve response times and even provide offline experiences
- Data no longer lives on one server but is distributed
- Each server may only have a piece of the data
- Server's need to be predictive and resilient
- Browser is orchestrating lots of asynchronous activity
- New I/O - voice, location, motion, VR, AR



Real example: pkglink

Created project to find duplicate npm packages and create hard links between the files to save space.

- https://github.com/jeffbski/pkglink
- Challenges
  - Large FS tree to check
  - Many I/O ops necessary to verify package is an acceptable candidate for linking (version, modified dates of files, file sizes)
  - Good feedback, user can cancel at any time

Demo async search

Arrays

```
const arr = [10, 20, 30];
arr.forEach(x => console.log(x)); // 10, 20, 30
const bar = arr.filter(x => x < 25); // [10, 20]
const cat = arr.map(x => x * 10); // [100, 200, 300]

// chaining them together
arr
  .filter(x => x < 25)
  .map(x => x * 10)
  .forEach(x => console.log(x));
```

The Big Idea

Observables

What if we could operate on data in a similar composable fashion but it could arrive over time?

Observables (vs Promises)

- Zero to many values over time
- Cancellable
- Synchronous or Asynchronous
- Composable streams
- Observables can wait for subscribe to start
- RxJS provides large set of operators

Observables

```
const ob$ = Rx.Observable.create(obs => {
  obs.next(10); // send first value
  obs.next(20); // 2nd
  obs.next(30); // 3rd
  obs.complete(); // we are done
});

ob$.subscribe(x => console.log(x));
// 10, 20, 30
```

demo

Observable Errors

```
const ob$ = Rx.Observable.create(obs => {
  obs.next(10); // send first value
  obs.next(20); // 2nd
  obs.error(new Error('something bad'));
});

ob$.subscribe(
 x => console.log(x),           // next
 err => console.error(err),     // errored
 () => console.log('complete') // complete
);
```

demo

ob$.subscribe()

Two forms are supported: separate fn args, or passing an object.

```
ob$.subscribe(
 x => console.log(x),           // next
 err => console.error(err),     // errored
 () => console.log('complete') // complete
);

// OR using object
ob$.subscribe({
 next: x => console.log(x),               // next
 error: err => console.error(err),        // errored
 complete: () => console.log('complete') // complete
);
```

You don't have to define all, just subscribe to what you care about.

Observable.of()

Simple way to create an observable with a known number of values.

```
// emits two values: foo, bar and then completes
const ob$ = Rx.Observable.of('foo', 'bar');

ob$.subscribe({
  next: x => console.log(x),
  error: err => console.error(err),
  complete: () => console.log('complete')
});

// foo, bar, complete
```

demo marbles

ob$.do()

For side effects, still needs subscribe to exec

```
const ob$ = Rx.Observable.of('foo', 'bar')
  .do(
    x => console.log(x),
    err => console.error(err),
    () => console.log('complete')
  );

ob$.subscribe();
// foo, bar, complete
```

demo

Observable.throw()

Create an observable which raises an error.

```
const ob$ = Rx.Observable.throw(new Error('my error'));

ob$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});

// error my error
```

demo

Observable.from()

Create an observable from an array, promise, or another observable.

```
const prom = new Promise((resolve, reject) => {
  resolve('foo');
  /* or reject(new Error('my error')) */
});

const ob$ = Rx.Observable.from(prom);

ob$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});

// next foo
// complete
```

demo marbles

Observable.interval()

```
const int$ = Rx.Observable.interval(1000);

int$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});

// next 0
// next 1
// ...
```

demo marbles

Observable.timer() - single

```
// when is delay in ms or absolute Date
const int$ = Rx.Observable.timer(when);

int$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});

// next 0
// complete
```

demo

Observable.timer() - multi

Similar to interval but allows an initial delay first

```
// delay and delayBetween in ms
// repeats every delayBetween milliseconds
const int$ = Rx.Observable.timer(delay, delayBetween);

int$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});

// next 0
// next 1
// ...
```

demo marbles

ob$.timestamp()

```
const int$ = Rx.Observable.interval(1000)
  .timestamp();

int$.subscribe({
  next: x => console.log('next', x.value, x.timestamp),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});
// next 0 1378690776351
// next 1 1378690777313
// ...
```

demo

ob$.take(N)

```
const int$ = Rx.Observable.interval(1000)
  .take(2);

int$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});
// next 0
// next 1
// complete
```

demo marbles

ob$.debounceTime()

```
const int$ = Rx.Observable.interval(100)
  .take(5)
  .debounceTime(200);

int$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});
// next 4
// complete
```

demo marbles

ob$.throttleTime()

```
const int$ = Rx.Observable.interval(100)
  .take(5)
  .throttleTime(200);

int$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});
// next 0
// next 3
// complete
```

demo marbles

ob$.filter()

```
const int$ = Rx.Observable.interval(1000)
  .take(5)
  .filter(x => x % 2);

int$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});
// next 1
// next 3
// complete
```

demo marbles

ob$.map()

```
const int$ = Rx.Observable.interval(1000)
  .take(5)
  .map(x => `${x} banana`);

int$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});
// next 0 banana
// next 1 banana
// next 2 banana
// next 3 banana
// next 4 banana
// complete
```

demo marbles

chaining

```
const int$ = Rx.Observable.interval(1000)
  .take(5)
  .filter(x => x % 2)
  .map(x => `${x} banana`);

int$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});
// next 1 banana
// next 3 banana
// complete
```

demo

Observable.merge()

```
const ob$ = Rx.Observable.merge(
  Rx.Observable.interval(1000)
    .map(x => `${x}s*****`),
  Rx.Observable.interval(200)
);

ob$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});
// next 0
// next 1
// next 2
// next 3
```

demo marbles

.combineLatest()

```
const a$ = Rx.Observable.interval(2000).map(x => `${x}s`);
const b$ = Rx.Observable.interval(1200);
const ob$ = Rx.Observable.combineLatest(
  a$,
  b$,
  (a, b) => ({
    a: a,
    b: b
  })
);

ob$.subscribe(x => console.log('next', x.a, x.b));
// next 0s 3
// next 0s 4
// next 0s 5
```

demo marbles

ob$.catch()

```
const ob$ = Rx.Observable.throw(new Error('my error'))
  .catch(err => Rx.Observable.of({ type: 'UNCAUGHT',
    payload: err,
    error: true }));

ob$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});
// next Object: {type: 'UNCAUGHT', payload: 'Error: my error at...'}
// complete
```

demo

Observable.ajax

```
const ob$ = Rx.Observable.ajax.getJSON('https://reqres.in/api/users')
  .map(payload => payload.data); /* use data prop */

ob$.subscribe({
  next: x => console.log('next', JSON.stringify(x, null, 2)),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});
// next [{ id: 1, first_name: 'george'...
//    { id: 2, first_name: 'lucille'...
//    ...
// complete
```

demo

ob$.mergeMap

```
Rx.Observable.of('redux', 'rxjs')
  .mergeMap(x => Rx.Observable.ajax({
      url: `https:npmsearch.com/query?q=${x}&fields=name,description`,
      crossDomain: true,
      responseType: 'json'
  })
  .map(ret => ret.response.results))  /* use results prop of payload */
  .subscribe({
    next: x => console.log('next', JSON.stringify(x, null, 2)),
    error: err => console.log('error', err),
    complete: () => console.log('complete')
  });
// next [{ name: 'react-redux-confirm-modal' ... ]
// next [{ name: 'rxjs-serial-subscription' ... ]
// complete
```

demo

Subject

```
const sub$ = new Rx.Subject();
sub$.next(10);

sub$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});

sub$.next(20);
sub$.next(30);
sub$.complete();
// next 20
// next 30
// complete
```

demo

Observable.webSocket()

```
const wsSubject = Rx.Observable.webSocket({
  url: 'ws://localhost:8010',
  // WebSocketCtor: WebSocket, // only for Node.js, import WebSocket from 'ws';
  resultSelector: x => x.data  // default is JSON.parse(x.data)
});

wsSubject.next('foo'); // queue foo for sending
wsSubject.next('bar'); // queue bar for sending

// connect to websocket, send queued msgs, listen for responses
wsSubject.subscribe(
  x => console.log('received', x),
  err => console.error('error', err),
  () => console.log('done')
);
```

demo

Reconnecting websocket

```
const wsSubject = Rx.Observable.webSocket({
  url: 'ws://localhost:8010',
  // WebSocketCtor: WebSocket, // only for Node.js, import WebSocket from 'ws';
  resultSelector: x => x.data});  // default is JSON.parse(x.data)

const reconWS$ = wsSubject
  .retryWhen(errors =>
    errors
    .do(err => console.error(err))
    .switchMap(err => Rx.Observable.timer(1000)));

setInterval(() => wsSubject.next(Date.now()), 1000);

// connect to websocket, send msgs, listen for responses
reconWS$.subscribe(x => console.log('received', x));
```

demo

redux-logic

```
const npmSearchLogic = createLogic({
  type: NPM_SEARCH,
  debounce: 500, // ms
  latest: true, // take latest only
  processOptions: {
    successType: searchFulfilled, // action creator to wrap success result
    failType: searchRejected      // action creator to wrap failed result
  },
  process({ getState, action }) {
    return Rx.Observable.ajax({
      url: `https://npmsearch.com/query?q=${action.payload}&fields=name,description`
      crossDomain: true,
      responseType: 'json'
    }).map(ret => ret.response.results)};  // use results prop of payload
});
```

Learning more

- http://reactivex.io/rxjs/manual/tutorial.html
- http://reactivex.io/rxjs/
- https://codewinds.com/

Summary

- RxJS helps you manage events and data over time
- It lets deal with data in functional steps
- Don't feel you have to learn it all, pick it up over time

Thanks

- https://codewinds.com/mwjs2017 (slides, resources)
- https://codewinds.com/ (newsletter tips/training)
- jeff@codewinds.com
- @jeffbski