RxJS Fundamentals

A **Frontend Masters** Workshop

Steve Kinney, @stevekinney — Frontend Architect at Temporal, https://temporal.io

Agenda

What are we going to do together?

- Get a conceptual understanding what an observable even is and what kind of problems you could possibly consider solving with them.
- Get our hands dirty by playing around with some simple test cases.
- Build a few really simple applications using observables.
- Solve for some common UX issues using observables.

What? Why?

Think of RxJS as Lodash for events.

—The RxJS Documentation

A promise is an eventual value.

An observable is a series of eventual values.

Working with APIs

What happens when things go wrong?

- Promises work great for the happy path.
- But, what if you want to retry a certain number of times?
- What happens if you want to reload the data at some regular interval.
- What happens if you want to iterate across a paginated API, but also show the data as it comes in?
- Observables can help with this.

Progressive Data Enhancement

Coordinating multiple API requests

- You might have come across the issue where you have multiple APIs that you need to coordinate in order to display a given piece of UI.
- This can be complicated and error-prone.
- Observables can help.

Loading Indicators

Figuring out when to show a spinner can be hard.

- If an API request is taking too long, it can be helpful to give our users from visual feedback.
- But, this can be tricky.
- What if it’s fast? We don’t want to immediately flash the spinner only to remove it 5 milliseconds later.
- Observables can help you deal with events that happen over time easily.

Dragging and Dropping

Manage adding and removing event listeners with ease.

- It’s not just APIs.
- Implementing a drag-and-drop interface can come with its own set of challenges.
- You want to start listening to mouse movements when the user clicks on the elements.
- But you want to stop as soon as they release the mouse.
- Other objects might case about where that object is—but only when it’s moving.

How?

of

Create an observable out of one or more values.

```
const example$ = of(1, 2, 3);

example$.subscribe(val => console.log(val));
```

from

Creates an observable from a promise or anything array- or observable-like.

```
const example$ = from([1, 2, 3]);

example$.subscribe(val => console.log(val));
```

from

Generator functions are kind of like observables, right?

```
const example$ = from(fibonacci(10));

example$.subscribe(val => console.log(val));
```

from

Promises are kind of like observables, right?

```
const example$ = from(Promise.resolve(1));

example$.subscribe(val => console.log(val));
```

What's the difference?

When to use from versus of.

- Use from when whatever you're working with is *kind of like* an observable.
  - Arrays.
  - Promises.
  - Objects with a subscribe method (a.k.a. have a similar API like Svelte Stores).

- Use of when it's *nothing* like an observable.
  - Primitives: strings, numbers, booleans, null, undefined.
  - Objects.

```
example$.subscribe(
  /* First Argument: Next */
  (value) => {
    console.log(
      'The first function is called whenever a value is emitted.',
      value
    );
  },
  /* Second Argument: Error */
  (error) => {
    console.error(
      'The second function is called if an error shows up in the observable.',
      error
    );
  },
  /* Third Argument: Complete */
  () => {
    console.log(
      'The third function is called whenever the observable completes.'
    );
  }
);
```

```
example$.subscribe(
  /* First Argument: Next */
  (value) => {
    console.log(
      'The first function is called whenever a value is emitted.',
      value
    );
  },
  /* Second Argument: Error */
  null,
  /* Third Argument: Complete */
  () => {
    console.log(
      'The third function is called whenever the observable completes.'
    );
  }
);
```

```
example$.subscribe(
  /* First Argument: Next */
  null,
  /* Second Argument: Error */
  null,
  /* Third Argument: Complete */
  () => {
    console.log(
      'The third function is called whenever the observable completes.'
    );
  }
);
```

Not only is this gross. It’s also deprecated.

Two Ways of Listenings to Observables

You'll probably use the first one like 90% of the time.

```
// Use this when you care only about the next values.
// This is the one that you'll use most of the time.
example$.subscribe((value) => console.log('Emitted', value));

// Use this when you explicitly care about either the error or when the
// observable completes.

example$.subscribe({
  next: (value) => console.log('Emitted', value),
  error: (error) => console.error('Error', error),
  complete: () => console.log('Complete')
});
```

Exercise

Creating Observables

We have a set of tests in ***exercises/creating-observables.test.js***. You can run *just* these tests using the the following.

npm test creating

Your mission: un-skip each test and make sure they pass. I'll do a few in ***exercises/basic-observables.test.js*** to get you started.

fromEvent

Create observables based on event listeners.

```
const button = document.querySelector('button');

button.addEventListener('click', (event) => {
  console.log(event);
});

const buttonClicks$ = fromEvent(button, 'click');

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

fromEvent

Optionally, you can pass a function to format the event.

```
fromEvent(input, 'input', (event) => {
  event.target.value;
});
```

bindCallback

Turn anything that takes a callback into an observable.

```
const get = bindCallback(jQuery.get);
const data$ = get('/api/endpoint');

const readFile = bindNodeCallback(fs.readFile);
const content$ = readFile('./groceries.md', 'utf-8');
```

fromFetch

Wrap the Fetch API in an observable.

```
import { fromFetch } from ‘rxjs/fetch';

const data$ = fromFetch('/api/endpoint');
```

Exercise

Using fromEvent

In ***applications/from-event,*** we have an incredibly basic example. Can you use fromEvent as an alternative to an event listener?

- Use fromEvent to create an observable that streams click events.
- Subscribe to that observable.
- Use addMessageToDOM to add a useless message to the DOM whenever the stream emits a value.

interval

Creates an observable that emits an event every n milliseconds.

```
const { interval } = require('rxjs');

const startingTime = Date.now();
const tick$ = interval(1000);

tick$.subscribe(() => console.log(Date.now() - startingTime));

// Logs:  1002, 2002, 3002, 4003, 5002
```

timer

Create an observable that emits after a certain amount of time.

```
const { timer } = require("rxjs");

const startingTime = Date.now();
const tick$ = timer(5000);

tick$.subscribe(() => console.log(Date.now() - startingTime));

// Logs: 5002
```

timer

Timers can also continue to emit at regular intervals.

```
const { timer } = require('rxjs');

const startingTime = Date.now();
const tick$ = timer(2000, 5000);

tick$.subscribe(() => console.log(Date.now() - startingTime));

// Logs:  2003, 7007, 12006
```

Unsubscribing

You should be able to clean up after yourself.

```
const interval$ = interval(1000);

const subscription = interval$.subscribe(console.log);

setTimeout(() => subscription.unsubscribe(), 5000);
```

Exercise

Increment a Counter at an Interval

We know how to create observables that fire at regular intervals using timer and interval.

We know how to create observables from DOM events using `fromEvent` and listen to them.

Given the *very* simple UI in ***applications/basic-counter***, can you wire up this simple counter.

Operators

```
export function* fibonacci() {
  let values = [0, 1];

  while (true) {
    let [current, next] = values;

    yield current;

    values = [next, current + next];
  }
}
```

```
export function* fibonacci(iterations) {
  let iteration = 0;
  let values = [0, 1];

  while (iteration < iterations) {
    let [current, next] = values;

    yield current;

    values = [next, current + next];
    iteration++;
  }
}

fibonacci(10);
```

Manipulating the Stream

Each observable has a .pipe method.

- This method takes one or more functions called *operators*.
- Each operator takes the observable, does something to it, and returns a new observable.
- This is similar to method chaining.
- Or, just using pipe in Lodash.

take

Take a certain number of values from an observable and then stop.

```
const example$ = from(fibonacci()).pipe(take(10));

example$.subscribe((val) => console.log(val));
// Logs: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

skip

Ignore the first however many values.

```
const example$ = from([1,2,3,4,5]).pipe(skip(2));

example$.subscribe((val) => console.log(val));
// Logs: 3, 4, 5
```

takeWhile

skipWhile

Take or skip values as long a certain condition is met. Complete the observable when the condition returns false.

```
const under200$ = from(fibonacci()).pipe(
  takeWhile((value) => value < 200)
);

under200$.subscribe(console.log);

// Logs: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

const over100$ = from(fibonacci()).pipe(
  skipWhile((value) => value < 100),
  take(4)
);

over100$.subscribe(console.log);

// Logs:  144, 233, 377, 610
```

Pop Quiz

Can anyone tell me why this will never emit an event?

```
const example$ = from(fibonacci()).pipe(
  skipWhile((value) => value < 100),
  takeWhile((value) => value > 500)
);

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

There are over 100 operators.

filter

This works just like it does on arrays.

```
const evenNumbers$ = of(1,2,3,4,5,6,7,8).pipe(
  filter((n) => n % 2 === 0)
);

evenNumbers$.subscribe((val) => console.log(val));
// Logs: 2, 4, 6, 8
```

map

You probably could have guessed this one.

```
const doubledNumbers$ = of(1, 2, 3).pipe(map((n) => n * 2));

doubledNumbers$.subscribe(console.log);
// Logs: 2, 4, 6
```

mapTo

This is just a simplified version of map.

```
const over100$ = from(fibonacci()).pipe(
  skipWhile((value) => value < 100),
  take(4),
  mapTo('HELLO!')
);

over100$.subscribe(console.log);

// Logs: "HELLO!", "HELLO!", "HELLO!", "HELLO!"
```

reduce

A little less useful than the next one, but it also shares a familiar name.

```
const under200$ = from(fibonacci()).pipe(
  takeWhile((value) => value < 200),
  reduce((total, value) => total + value, 0)
);

under200$.subscribe(console.log);

// Logs: 375
```

scan

Like reduce, but more useful for our purposes.

```
const under200$ = from(fibonacci()).pipe(
  takeWhile((value) => value < 200),
  reduce((total, value) => total + value, 0)
);

under200$.subscribe(console.log);

// Logs: 1, 3, 6, 11, 19, 32, 53, 87, 142, 231, 375
```

scan

This means, that we could do something like this.

```
const fibonacci$ = range(0, Infinity).pipe(
  scan((total, value) => total + value, 0),
  take(4),
);

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

tap

Incredibly useful for side-effects like DOM manipulation and debugging.

```
const div = document.querySelector('div');

const example$ = from([1, 2, 3, 4]).pipe(
  tap((value) => console.log(`About to set the <div> to ${value}.`)),
  tap((value) => {
    div.innerText = value;
  }),
  tap((value) => console.log(`Set the <div> to ${value}.`))
);
```

Exercise

Basic Operators

Sure, I could make you endure a tour through all of the operators, or we *could* just get our hands dirty and take some of them for a spin.

You can find the tests in ***exercises/basic-operators.test.js***.

takeUntil

skipUntil

These operators rely on other observables. When the observable that *they're* subscribing to triggers, they will activate.

```
const startingTime = Date.now();

const firstTimer$ = timer(2000);
const secondTimer$ = timer(7000);

const example$ = interval(1000).pipe(
  skipUntil(firstTimer$),
  takeUntil(secondTimer$),
);

example$.subscribe(() => console.log(Date.now() - startingTime));

// Logs: 2004, 3000, 4000, 5001, 6002
```

Exercise

Improving Our Counter

When last we checked on our Basic Counter, it left a lot to be desired.

But, now we can start and stop an observable based on other observables.

Can you get it working with the operators to the left?

- takeUntil
- skipUntil
- scan

Manipulating Time

delay

Delay each emitted value for a given number of milliseconds.

```
const startingTime = Date.now();
const elapsed = () =>  Date.now() - startingTime;

const example$ = of(1, 2, 3, 4).pipe(
  delay(1000)
);

example$.subscribe((value) => {
  console.log({
    value,
    emittedAt: elapsed()
  });
});
```

```
// {value: 1, emittedAt: 1003}
// {value: 2, emittedAt: 1004}
// {value: 3, emittedAt: 1004}
// {value: 4, emittedAt: 1004}
```

debounceTime

Ignores events until a certain amount of time has passed since the last event.

```
const startingTime = Date.now();
const elapsed = () => Date.now() - startingTime;

const example$ = of(1, 2, 3, 4).pipe(debounce(1000));

example$.subscribe((value) => {
  console.log({
    value,
    emittedAt: elapsed()
  });
});

// Logs: {value: 4, emittedAt: 1}
```

throttleTime

After it emits a value, it starts ignoring values until a given amount of time has passed.

```
const startingTime = Date.now();
const elapsed = () => Date.now() - startingTime;

const example$ = of(1, 2, 3, 4).pipe(throttleTime(1000));

example$.subscribe((value) => {
  console.log({
    value,
    emittedAt: elapsed()
  });
});

// Logs: {value: 1, emittedAt: 1}
```

Merging Timelines

Combining Streams

We have a few different strategies.

- merge: Combine multiple streams, anytime one of them emits a value, emit that value.
- concat: Work your way through the first stream before moving on to the next stream.
- forkJoin: Wait for everything to complete and then give me the last value that come out of each observable.
- race: Go with whatever one emits first and ignore the rest.

merge

Combine multiple streams into one.

```
const firstStream$ = of(1, 2, 3).pipe(delay(1000));
const secondStream$ = of('a', 'b', 'c');

const merged$ = merge(firstStream$, secondStream$);

merged$.subscribe(console.log);

// Logs: a, b, c, 1, 2, 3
```

concat

Play through the first observable, then move on to the next.

```
const firstStream$ = of(1, 2, 3).pipe(delay(1000));
const secondStream$ = of('a', 'b', 'c');

const concatenated$ = concat(firstStream$, secondStream$);

concatenated$.subscribe(console.log);

// Logs: 1, 2, 3, a, b, c
```

forkJoin

Wait for all of the observables to complete and then hand us the last value emitted from each.

```
const firstStream$ = of(1, 2, 3).pipe(delay(1000));
const secondStream$ = of('a', 'b', 'c');

const joined$ = forkJoin(firstStream$, secondStream$);

joined$.subscribe(console.log);

// Logs: [3, "c"]
```

race

Kind of like Promise.race(), this listens for the first observable to emit a value and then ignores any of the other ones.

```
const firstStream$ = of(1, 2, 3).pipe(delay(1000));
const secondStream$ = of('a', 'b', 'c');

const winnersOnly$ = race(firstStream$, secondStream$);

winnersOnly$.subscribe(console.log);

// Logs: a, b, c
```

partition

Bonus: This one *technically* doesn't fit in with the rest as it does the opposite. It takes one stream and splits it into two observables based on a function.

```
const example$ = of(1, 2, 3, 4, 5, 6);
const [evens$, odds$] = partition(
  example$,
  (value) => value % 2 === 0
);

evens$.subscribe((x) => console.log('evens', x));
odds$.subscribe((x) => console.log('odds', x));
```

Follow Along

Experimenting with Combinations

Seeing some of this visually might be helpful.

Let's pop over to ***applications/merging-timelines*** and play around a bit with some of the different ways in which we can combine observables.

Higher-Order Observables

As with most things in programming, everything is easy until you start nesting things.

Returning Observables

What happens when map returns an observable instead of a value?

```
const example$ = of(1, 2, 3, 4).pipe(
  map((value) => of(value).pipe(
    delay(value * 1000)
  ))
);

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

Returning Observables

This is not what we wanted.

```
// Observable { … }
// Observable { … }
// Observable { … }
// Observable { … }
```

Returning Observables

This doesn’t work either.

```
const example$ = merge(
  of(1, 2, 3, 4).pipe(
    map((value) => of(value).pipe(
      delay(value * 1000)
    )
  ))
);

// Observable { … }
// Observable { … }
// Observable { … }
// Observable { … }
```

Returning Observables

This might seem esoteric, but it’s more common than you think.

```
const results$ = fromEvent(button, 'click').pipe(
  map(() => fromFetch('/api/important-stuff'))
);
```

mergeAll

Takes a stream of branch observables and merges them into the main timeline.

```
const example$ = of(1, 2, 3, 4).pipe(
  map((value) => of(value).pipe(
    delay(value * 1000)
  )),
  mergeAll()
);

example$.subscribe((value) => {
  console.log({
    value,
    elapsedTime: Date.now() - startingTime
  });
});
```

```
// {value: 1, elapsedTime: 1006}
// {value: 2, elapsedTime: 2002}
// {value: 3, elapsedTime: 3006}
// {value: 4, elapsedTime: 4002}
```

concatAll

Similar to mergeAll and mergeMap, but it plays through the observables in order.

concatMap

```
const example$ = of('a', 'b', 'c', 'd').pipe(
  map((letter) =>
    interval(1000).pipe(
      map((i) => letter + ' ' + i),
      take(4)
    )
  ),
  concatAll()
);

const example$ = of('a', 'b', 'c', 'd').pipe(
  concatMap((letter) =>
    interval(1000).pipe(
      map((i) => letter + ' ' + i),
      take(4)
    )
  ),
);
```

Mapping Operators

We have four major kinds of mapping operators.

- mergeMap: Just like merge. Combine all of the children in to one stream.
- concatMap: Combine all of the children in one stream, but work through them one at a time.
- switchMap: Drop what you're doing when a new child emits for the first time.
- exhaustMap: Start working on the first child that emits, but then ignore everything else until what you're working on completes.

combineLatestAll

Wait for them all to emit at least once, and then any time one of them emits, give us an array with all of the latest values.

```
const example$ = of('a', 'b', 'c', 'd').pipe(
  map((letter) =>
    interval(1000).pipe(
      map((i) => letter + ' ' + i),
      take(4)
    )
  ),
  combineLatestAll()
);
```

combineLatestWith

Wait for them all to emit at least once, and then any time one of them emits, give us an array with all of the latest values.

```
const firstInputChanges$ = fromEvent(firstInput, 'change');
const secondInputChanges$ = fromEvent(secondInput, 'change');

firstInputChanges$.pipe(
  combineLatestWith(secondInputChanges$),
  map(
    ([first, second]) =>
      Number(first.target.value) + Number(second.target.value)
  )
);
```

NEVER

A built-in stream that *never* emits and it *never* completes.

```
const never$ = NEVER;
```

EMPTY

An observable that immediately completes. That’s it.

```
const empty$ = EMPTY;
```

Summary and Questions