RxJS and Ajax

What are we going to cover?

RxJS Observables

RxJS Operators

RxJS Subjects

Doing HTTP requests with RxJS

RxJS

RxJS is an API for **asynchronous** programming with **observable streams**

Why RxJS?

Most actions are **not standalone** occurrences

- Example: A mouse click triggers an Ajax request which triggers a UI update

RxJS is a great library to compose these streams in a **functional style**

The RxJS Observable

An **Observable** is the object that emits a stream of event

- The observer is the code that subscribes to the event stream

RxJS operators

Operators are used to operate on the event stream between the source and the subscriber

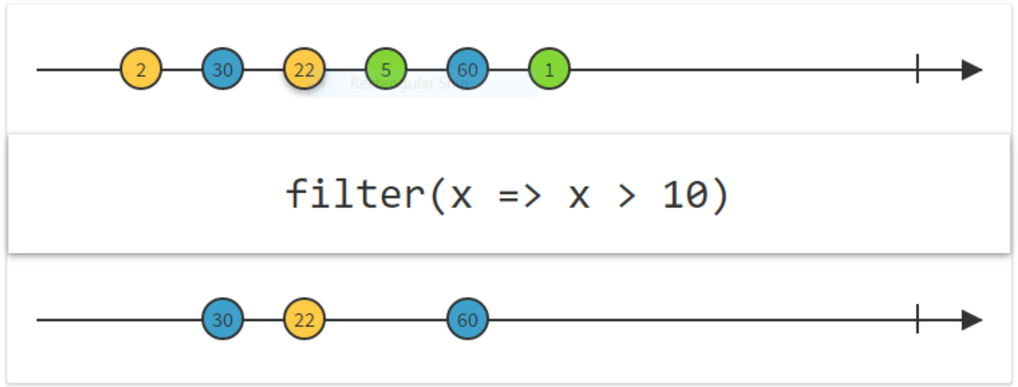
There are **many operators** for all sorts of purposes:

- Creating observables
- Transforming
- Filtering
- Combining
- Error handling
- Aggregate
- …

Example: The filter operator

An observable **filter** only allows items through if they pass the filter.

- These are normally documented using a before and after observable timeline



```
filter(x => x > 10)
```

Basic RxJS example

```
import { range } from "rxjs"
import { filter, map } from "rxjs/operators"


range(1, 10)
◦ .pipe(
    // 1 to 10
    filter(x => x % 2 === 0),
    // Only even numbers
    map(x => x * 10)
    // Multiplied by 10
  )
  .subscribe(x => console.log(x));
```

Combining RxJS streams

```
// Start with 10 numbers
range(1, 10)
  .pipe(
    // Switch to promises
    concatMap(page => fetch(`/api/movies?page=${page}`)),
    // Get the result per page using a promise
    flatMap(rsp => rsp.json()),
    // Switch to the resulting movies array per page
    map(json => json.results),
    // Convert stream of arrays to stream of movies
    flatMap(e => e)
  )
  // Print each movie object
  .subscribe(movie => console.log(movie));
```

Subject

A **Subject** is both an **Observable** and an **Observer**

- Useful for when you want to emit values

There are a number of different Subject types with very specific goals

- ReplaySubject
  - Resubmits all previously submitted values
- BehaviorSubject
  - Resubmits the last submitted value
- AsyncSubject
  - Submits only the last value when the stream completes

Ajax and RxJS

Do any **AJAX** request you want and subscribe to the result

Ajax and RxJS

```
import { ajax } from "rxjs/ajax";

ajax
  .getJSON("/api/movies")
  .subscribe(movies => console.table(movies));
```

Retrying failed requests

The **retry()** and **retryWhen()** operators make it easy to retry failed attempts

- Works on any Observable, not just Ajax requests

The **retry()** operator retries **immediately** for the specified number of times

- Can result in a busy server to become overloaded

The **retryWhen()** operator retries when the returned **stream emits**

- Allows for waiting before retrying

Retrying failed requests

```
ajax
  .getJSON('/api/movies')
  .pipe(
    retryWhen(error$ =>
      error$.pipe(
        map(() => 100),
        scan((p, c) => p + c),
        delayWhen(wait => timer(wait)),
        take(5)
      )
    )
  )
  .subscribe(movies => console.table(movies));
```

A Redux like observable Store

A basic Redux like **store** is easy to implement using RxJS

- Use the scan() operator to reduce dispatched actions

Use a **higher order component** to listen for store changes

Write **reducers** just like you would for Redux

A RxJS store example

```
const action$ = new Subject()
export const dispatch = action => action$.next(action);

export const store$ = action$.pipe(
  startWith({
    type: '__INIT__'
  }),
  scan(reducer, {}),
  shareReplay(1)
);
```

The HOC connect

```
import { store$ } from "./store";
export const connect = mapStateToProps => WrappedComponent => {
  return class extends Component {
    state = {};
    subscription = null;
    componentDidMount() {
      this.subscription = store$.subscribe(state =>
        this.setState(mapStateToProps(state)));
    }
    componentWillUnmount() {
      this.subscription.unsubscribe();
    }
    render() {
      return <WrappedComponent {...this.props} {...this.state} />;
    }
  };
};
```

Conclusion

RxJS Observables are very powerful

- They are much more powerful than promises

Most of the power in RxJS comes from all the operators

- Manipulate, combine or stop streams as needed

RxJS Subjects alow for a lot of control

- Both Observer and Observable

Doing AJAX requests with RxJS is easy