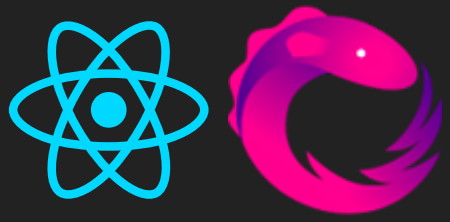
REACT AND RXJS



React Rally 2015-08

Seth House @whiteinge

OBSERVABLES

- A unified API for async operations:
    - DOM events.
    - Ajax.
    - SSE & websockets.
    - setInterval.
- A stream of events.

WHY REACT?

- Performant & simple API.
- Functional friendly.
- Mostly declarative.
- Flux.

WHY REACTIVE EXTENSIONS?

- Performant & mature API.
- Functional.
- Declarative.

OBSERVERS

Make an observer (if you're a library author):

```
var Rx = require('rx');

var myobsv = Rx.Observer.create(
    (x) => console.log('onNext', x),
    (err) => console.log('onError', err),
    () => console.log('onCompleted'));

myobsv.onNext(1);
myobsv.onNext(2);
myobsv.onCompleted();
```

OBSERVERS

Get a pre-made observer (if you're everyone else):

```
var myelem = document.querySelector('someel.myelem');
var clicks = Rx.Observable.fromEvent(myelem, 'click')
```

OBSERVERS

Pre-made observers for everything:

```
Rx.Observable.from(...)
Rx.Observable.fromEvent(...)
Rx.Observable.fromEventPattern(...)
Rx.Observable.fromCallback(...)
Rx.Observable.fromNodeCallback(...)
Rx.Observable.fromPromise(...)
Rx.Observable.generate(...)
Rx.Observable.generateWithAbsoluteTime(...)
Rx.Observable.generateWithRelativeTime(...)
```

OBSERVABLES

“Think of an Observable as an asynchronous immutable array.”

```
var mysubscription = myobsv.subscribe(
    (x) => console.log('onNext', x),
    (err) => console.log('onError', err),
    () => console.log('onCompleted'));
```

OBSERVABLES

Stop Listening:

```
mysubscription.dispose();
```

OBSERVABLES

Stop Listening Automatically

```
var myelem = document.querySelector('someel.myelem');
var clicks = Rx.Observable
    .fromEvent(myelem, 'clicks')
    .take(1);
```

OBSERVABLE METHODS

- Filtering (`filter`)
- Transforming (`map`, `reduce`, `flatMap`)
- Collecting (`scan`)
- Buffering (`buffer`, `bufferWithCount`, `bufferWithTime`, `sample`, `debounce`)
- Combining (`merge`, `concat`, `combineLatest`)

AJAX EVENTS

Fetching usernames from GitHub:

```
var github_users = Rx.DOM.ajax({
        method: 'GET',
        url: 'https://api.github.com/users'});
    .filter(x => x.status === 200)
    .map(JSON.parse)
    // Cache the deserialized response.
    .shareReplay(1)
    // Explode items in JSON response into stream items.
    .flatMap(x => Rx.Observable.from(x));

var usernames_list = github_users
    .pluck('login')
    .subscribe(x => console.log('GitHub user:', x));
```

MERGE AJAX EVENTS WITH DOM EVENTS

Combine users with click events:

```
var clicks = Rx.DOM.click(document.querySelector('#thelink'));

// Combine each click with a user.
clicks.zip(usernames_list, (click, user) => user)
    // When the list is exhausted the event handler is removed.
    .subscribe(x => console.log('GitHub user:', x));
```

A unified API for async operations.

MERGE MULTIPLE AJAX REQUESTS

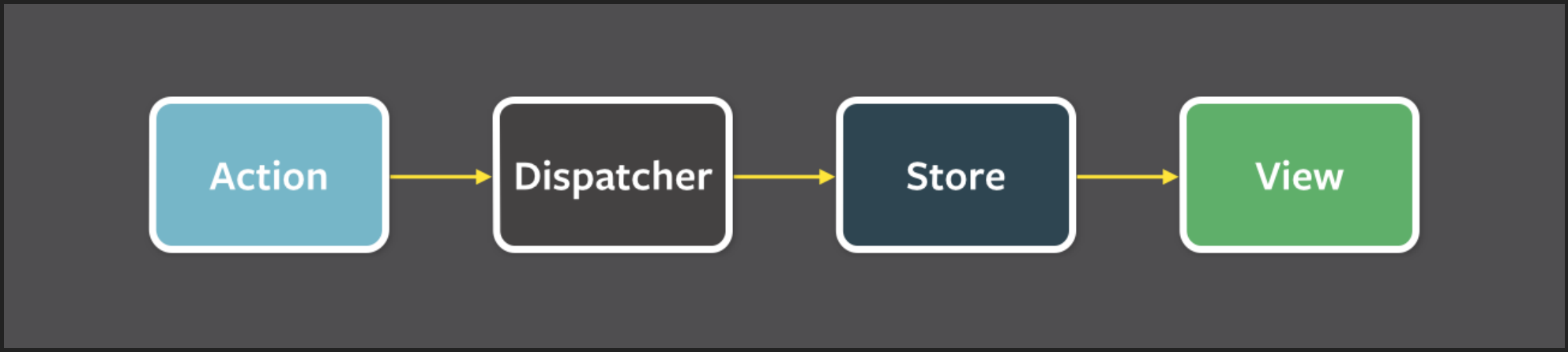
How do you want to combine the results?

- Output each response as soon as it comes in? (`merge`)
- Output each response in the same order as the request? (`concat`)
- Wait for both to complete and combine them? (`zip` & `flatMap`)

FLUX

A code organization pattern:

- Unidirectional data flow.
- Immutable data structures are helpful.
- State lives in predictable places.



FLUX VARIANTS

Alt, Barracks, Delorean, disto, fluce, fluctuations, Flummox, Flux, Flux This, Fluxette, Fluxible, Fluxxor, Fluxy, Lux, Marty.js, MartyJS, Material Flux, McFly, microcosm, microflux, mmox, Nuclear.js, NuclearJS, OmniscientJS, Reducer, Redux, Reflux

VARIATIONS ON A THEME

- Often kill / hide dispatcher.
- Reduce boilerplate.
- Async helpers.

RX

Shares the Flux philosophy:

- Unidirectional data flow.
- "Immutable" data structures.
- Minimization of state tracking.

DISPATCHER

Push messages to subscribers; receive messages from elsewhere:

```
var Dispatcher = new Rx.Subject();
```

DISPATCHER

Example message:

```
{
    channel: MY_MODULE,
    type: act.SOME_ACTION,
    data: {extraData: 'important'},
    args: [DOMEvent],
}
```

STORE

Subscribe only to certain channels:

```
var myStore = Dispatcher
    .filter(x => x.channel === channels.MY_MODULE);
```

STORE

Manage state; don't store state.

```
var myAjax = myStore
    .startWith({type: act.REFRESH})
    .filter(x => x.type === act.REFRESH)
    .flatMap(() => Rx.DOM.get('/some/url'))
    .shareReplay(1);

var myAjaxSummarized = myAjax
    .pluck('subdata')
    .map(x => {x.someval.toLower(); return x})
    .map(summarizeData);
```

STORE

Accumulate values over time:

```
var clickCounter = myStore
    .filter(x => x.type === act.CLICK)
    .scan((acc, x) => { acc += 1; return acc }, 0);
```

STORE

Combine multiple stores:

```
import {otherStore} from 'related/module';

var combinedStore = myStore
    .combineLatest(otherStore, (x, y) => ({x, y}));
```

VIEW

Just another transformation step in the stream.

```
var app = myAjaxSummarized
    .map(function(summaryData) {
        return (
            <h3 onClick={(ev) => {Dispatcher.onNext({
                channel: channels.MY_MODULE,
                type: act.REFRESH,
            })}}>Refresh</h3>

            <ul>
                {summaryData.map(x => <li>x</li>)}
            </ul>
        );
```

Push messages back into the Dispatcher. (Constants / action creators.)

HELPERS

Enforce an interface with a helper function.

```
function sendMsg(channel, action, data = {}) {
    return function(...args) {
        var msg = {channel, action, data, args};
        return Dispatcher.onNext(msg);
    };
}
```

Reduce repetition with currying.

```
const send = _.curry(sendMsg, channels.MY_MODULE, 2);

<h3 onClick={
    send(type.REFRESH, {extraData: 'important'})
}>Refresh</h3>
```

SIDE-EFFECTS

Rendering to the DOM is a one-way side-effect.

```
app.subscribe(function(content) {
    React.render(
        React.createElement('div', content),
        document.querySelector('#content'));
});
```

MORE SIDE-EFFECTS

Log the dispatcher:

```
Dispatcher.subscribe(function(event) {
    console.log('Dispatching event', event);
});
```

AND MORE SIDE-EFFECTS

Upload analytics information:

```
Dispatcher
    .map(formatAndFilterData)
    .bufferWithTimeOrCount(300, 100)
    .subscribe(
        uploadAnalyticsData,
        uploadAnalyticsErrors);
```

EVEN MORE SIDE-EFFECTS

Record the dispatcher:

```
Dispatcher
    .takeLast(100)
    .subscribe(writeRecording);

// => [
//  {type: act.AUTOCOMPLETE, data: {text: 's'}},
//  {type: act.AUTOCOMPLETE, data: {text: 'se'}},
//  {type: act.AUTOCOMPLETE, data: {text: 'sea'}},
//  {type: act.AUTOCOMPLETE, data: {text: 'sear'}},
//  {type: act.AUTOCOMPLETE, data: {text: 'searc'}},
//  ...
// ]
```

EVEN MORE SIDE-EFFECTS

Play back the recording:

```
var recording = readRecording();
var tgtGroupStore = Rx.Observable
    .from(recording)
    .zip(Rx.Observable.interval(500), x => x);
```

RESOURCES

Slides: https://github.com/whiteinge/presentations

- The big list of Rx methods
- The Rx Decision Tree
- Netflix's introduction to map/filter/reduce/etc
- RxMarbles