Utilising ImmutableJS for top performance

Let's roll up our sleeves and apply our learnings from ImmutableJS to our React/Redux weather app.

WE'LL COVER THE FOLLOWING

- `shouldComponentUpdate`
- Additional Material

As a short experiment, try putting a `console.log('RENDER PLOT')` into the `render` method of the `Plot` component:

```
class Plot extends React.Component {
    /* … */
    render() {
        console.log('RENDER PLOT');
    return (
      <div id="plot" ref="plot"></div>
    );
  }
}
```

Now try using the app for a bit, clicking around, request data for different cities. What you might notice is *that the* `Plot` *rerenders even if we only change the location field and the plot itself stays the exact same*! (Look at your browser console).

```
import React from 'react';
import './App.css';
import { connect } from 'react-redux';

import Plot from './Plot';
import {
  changeLocation,
  setData,
  setDates,
  setTemps,
  setSelectedDate,
  setSelectedTemp,
  fetchData
} from './actions';

class App extends React.Component {
  fetchData = (evt) => {
    evt.preventDefault();

    var location = encodeURIComponent(this.props.location);

    var urlPrefix = '/cors/http://api.openweathermap.org/data/2.5/forecast?q=';
    var urlSuffix = '&APPID=dbe69e56e7ee5f981d76c3e77bbb45c0&units=metric';
    var url = urlPrefix + location + urlSuffix;

    this.props.dispatch(fetchData(url));

  };

  onPlotClick = (data) => {
    if (data.points) {
      var number = data.points[0].pointNumber;
      this.props.dispatch(setSelectedDate(this.props.dates[number]));
      this.props.dispatch(setSelectedTemp(this.props.temps[number]))
    }
  };

  changeLocation = (evt) => {
    this.props.dispatch(changeLocation(evt.target.value));
  };

  render() {
    var currentTemp = 'not loaded yet';
    if (this.props.data.list) {
      currentTemp = this.props.data.list[0].main.temp;
    }
    return (
      <div>
        <h1>Weather</h1>
        <form onSubmit={this.fetchData}>
          <label>City, Country
            <input
              placeholder={"City, Country"}
              type="text"
              value={this.props.location}
              onChange={this.changeLocation}
            />
          </label>
        </form>
        {/*
          Render the current temperature and the forecast if we have data
          otherwise return null
        */}
        {(this.props.data.list) ? (
          <div>
            {/* Render the current temperature if no specific date is selected */}
            {(this.props.selected.temp) ? (
              <p>The temperature on { this.props.selected.date } will be { this.props.selecte
            ) : (
              <p>The current temperature is { currentTemp }°C!</p>
            )}
            <h2>Forecast</h2>
            <Plot
              xData={this.props.dates}
              yData={this.props.temps}
              onPlotClick={this.onPlotClick}
              type="scatter"
            />
          </div>
        ) : null}

      </div>
    );
  }
}

// Since we want to have the entire state anyway, we can simply return it as is!
function mapStateToProps(state) {
  return state.toJS();
}

export default connect(mapStateToProps)(App);
```

This is a react feature, react rerenders your entire app whenever something changes. This doesn’t necessarily have a massive performance impact on our current application, but it’ll definitely bite you in a production application! So, what can we do against that?

`shouldComponentUpdate`

React provides us with a nice lifecycle method called `shouldComponentUpdate` which allows us to regulate when our components should rerender. As an example, try putting this into your `Plot`:

```
class Plot extends React.Component {
    shouldComponentUpdate(nextProps) {
        return false;
    }
    /* … */
}
```

Now try loading some data and rendering a plot. What you see is that *the plot never renders*. This is because we’re basically telling react above that no matter what data comes into our component, it should never render the `Plot`! On the other hand, if we `return true` from there we’d have the default behaviour back, i.e. rerender whenever new data comes in.

```
import React from 'react';

class Plot extends React.Component {
  shouldComponentUpdate(nextProps) {
                return false;
```

```
      }

  drawPlot = () => {
    Plotly.newPlot('plot', [{
      x: this.props.xData,
      y: this.props.yData,
      type: this.props.type
    }], {
      margin: {
        t: 0, r: 0, l: 30
      },
      xaxis: {
        gridcolor: 'transparent'
      }
    }, {
      displayModeBar: false
    });

    document.getElementById('plot').on(
      'plotly_click', this.props.onPlotClick);
  }

  componentDidMount() {
    this.drawPlot();
  }

  componentDidUpdate() {
    this.drawPlot();
  }

  render() {
    console.log('RENDER PLOT');
    return (
      <div id="plot"></div>
    );
  }
}


export default Plot;
```

As I've hinted with the variable above, `shouldComponentUpdate` gets passed `nextProps`. This means, in theory, we could check if the props of the `Plot` have changed and only rerender if that happens, right? Something like this:

```
class Plot extends React.Component {
    shouldComponentUpdate(nextProps) {
        return this.props !== nextProps;
    }
    /* … */
}
```

Well, here we hit the problem we talked about above. (`{ twitter: '@mxstbr' } !== { twitter: '@mxstbr' }`) Those will always be different since they might have the same content, but they won't be the same object!

This is where ImmutableJS comes in, because while we could do a *deep comparison* of those two objects, it's a lot cheaper if we could just do this:

```
class Plot extends React.Component {
    shouldComponentUpdate(nextProps) {
        return !this.props.equals(nextProps);
    }
    /* … */
}
```

Let's try getting some immutable data to our `Plot`!

In our `mapStateToProps` function, instead of returning `state.toJS()` we should just return the immutable state. The problem is that redux expects the value we return from `mapStateToProps` to be a standard javascript object, and it'll throw an error if we just do `return state;` and nothing will work.

So let's return an object from `mapStateToProps` that has a `redux` field instead:

```
function mapStateToProps(state) {
  return {
        redux: state
    };
}
```

Then, in our `App` we now have access to `this.props.redux`! We can access properties in there with `this.props.redux.get` (and `getIn`), so let's replace all instances where we access the state with that.

Let's start from the top, in `fetchData`. There's only a single `this.props.location` in there, which we replace with `this.props.redux.get('location')`:

```
class App extends React.Component {
  fetchData = (evt) => {
    evt.preventDefault();

    var location = encodeURIComponent(this.props.redux.get('location'));

    /* … */
  };

  onPlotClick = (data) => {/* … */};

  changeLocation = (evt) => {/* … */};

  render() {/* … */}
}
```

We don't access the props at all in `onPlotClick` and `changeLocation`, so we can skip those!

In `render`, the first access is already a bit more difficult – we want to replace `this.props.data.list`… Do you remember how to do that?

…

With `getIn`! Like this:

```
class App extends React.Component {
    fetchData = (evt) => {/* … */};

    onPlotClick = (data) => {/* … */};

    changeLocation = (evt) => {/* … */};

    render() {
        var currentTemp = 'not loaded yet';
        if (this.props.redux.getIn(['data', 'list'])) {
            /* … */
        }
        return (/* … */);
    }
}
```

Now, for the next one (`this.props.data.list[0].main.temp`) you might think of writing `this.props.redux.getIn(['data', 'list'])[0].main.temp`, but the problem is that `this.props.redux.getIn(['data', 'list'])` is an immutable array too!

So, instead we can just further use `getIn`:

```
class App extends React.Component {
  fetchData = (evt) => {/* … */};
  onPlotClick = (data) => {/* … */};

  changeLocation = (evt) => {/* … */};

  render() {
    var currentTemp = 'not loaded yet';
    if (this.props.redux.getIn(['data', 'list'])) {
      currentTemp = this.props.redux.getIn(['data', 'list', '0', 'mai
n', 'temp']);
    }
    return (/* … */);
  }
}
```

Similarly, we are going to fix `onPlotClick`:

```
onPlotClick = (data) => {
    if (data.points) {
      var number = data.points[0].pointNumber;
      this.props.dispatch(setSelectedDate(this.props.redux.getIn(['date
s', number])));
      this.props.dispatch(setSelectedTemp(this.props.redux.getIn(['temp
s', number])))
    }
};
```

Now let’s put all of this in action (*It will still not work but you should try running it. Can you guess what’s the issue?*)

```
import React from 'react';
import './App.css';
import { connect } from 'react-redux';
import { fromJS } from 'immutable';

import Plot from './Plot';
import {
  changeLocation,
  setData,
  setDates,
  setTemps,
  setSelectedDate,
  setSelectedTemp,
  fetchData
} from './actions';

class App extends React.Component {
  fetchData = (evt) => {
    evt.preventDefault();
    var location = encodeURIComponent(this.props.redux.get('location'));

    var urlPrefix = '/cors/http://api.openweathermap.org/data/2.5/forecast?q=';

    var urlSuffix = '&APPID=dbe69e56e7ee5f981d76c3e77bbb45c0&units=metric';
    var url = urlPrefix + location + urlSuffix;

    this.props.dispatch(fetchData(url));

  };

  onPlotClick = (data) => {
    if (data.points) {
      var number = data.points[0].pointNumber;
      this.props.dispatch(setSelectedDate(this.props.redux.getIn(['dates', number])));
      this.props.dispatch(setSelectedTemp(this.props.redux.getIn(['temps', number])))
    }
  };

  changeLocation = (evt) => {
    this.props.dispatch(changeLocation(evt.target.value));
  };

  render() {
    var currentTemp = 'not loaded yet';
    if (this.props.redux.getIn(['data', 'list'])) {
      currentTemp = this.props.redux.getIn(['data', 'list', '0', 'main', 'temp']);
    }
    return (
      <div>
        <h1>Weather</h1>
        <form onSubmit={this.fetchData}>
          <label>I want to know the weather for
            <input
              placeholder={"City, Country"}
              type="text"
              value={this.props.redux.get('location')}
              onChange={this.changeLocation}
            />
          </label>
        </form>
        {/*
          Render the current temperature and the forecast if we have data
          otherwise return null
        */}
        {(this.props.redux.getIn(['data', 'list'])) ? (
          <div>
            {/* Render the current temperature if no specific date is selected */}
            { this.props.redux.getIn(['selected', 'temp']) ? (
              <p>The temperature on { this.props.redux.getIn(['selected', 'date'])} will be {
            ) : (
              <p>The current temperature is { currentTemp }°C!</p>
            )}
            <h2>Forecast</h2>
            <Plot
              xData={this.props.redux.get('dates')}
              yData={this.props.redux.get('temps')}
              onPlotClick={this.onPlotClick}
              type="scatter"
            />
          </div>
        ) : null}
      </div>
    );
  }
}

// Since we want to have the entire state anyway, we can simply return it as is!
function mapStateToProps(state) {
  return {
                redux: state
        };
}

export default connect(mapStateToProps)(App);
```

As you might've noticed, this doesn't work though, the Plot doesn't render. Why? Well, take a look at how we pass in the data:

```
<Plot
    xData={this.props.redux.get('dates')}
    yData={this.props.redux.get('temps')}
    onPlotClick={this.onPlotClick}
    type="scatter"
/>
```

Let's take a peek at the only method where we use `this.props.xData` and `this.props.yData` in our `Plot` component:

```
// Plot.js

class Plot extends React.Component {
  drawPlot = () => {
    Plotly.newPlot('plot', [{
      x: this.props.xData,
      y: this.props.yData,
      type: this.props.type
    }], {/* … */}, {/* … */});
    /* … */
  }

  componentDidMount() {/* … */}
  componentDidUpdate() {/* … */}
  render() {/* … */}
}
```

This is where `toJS` comes in! Let's do this:

```
// Plot.js
```

```
class Plot extends React.Component {

  drawPlot = () => {
    Plotly.newPlot('plot', [{
      x: this.props.xData.toJS(),
      y: this.props.yData.toJS(),
      type: this.props.type
    }], {/* … */}, {/* … */});
    /* … */
  }

  componentDidMount() {/* … */}
  componentDidUpdate() {/* … */}
  render() {/* … */}
}
```

And now everything works again!

```
import React from 'react';

class Plot extends React.Component {
  drawPlot = () => {
    Plotly.newPlot('plot', [{
      x: this.props.xData.toJS(),
      y: this.props.yData.toJS(),
      type: this.props.type
    }], {
      margin: {
        t: 0, r: 0, l: 30
      },
      xaxis: {
        gridcolor: 'transparent'
      }
    }, {
      displayModeBar: false
    });

    document.getElementById('plot').on(
      'plotly_click', this.props.onPlotClick);
  }

  componentDidMount() {
    this.drawPlot();
  }

  componentDidUpdate() {
    this.drawPlot();
  }

  render() {
    console.log('RENDER PLOT');
    return (
      <div id="plot"></div>
    );
  }
}


export default Plot;
```

It sort of works but are you looking at your console. Remember those frequent `RENDER PLOT` statements in your console. They are still there (*yes we haven't removed them as yet*) and they are getting printed with every click on the graph even when there's no new data to update the graph.

Hence, we still haven't solved the original problem though, the `Plot` still rerenders everytime something changes, even if it's not related to the Plot. Really, *the only time we ever want that component to rerender is when either `xData` or `yData` changes!*

Let's apply our knowledge of ImmutableJS and of `shouldComponentUpdate`, and fix this together. Let's check if `this.props.xData` and `this.props.yData` are the same and only rerender if one of them changed:

```
import React from 'react';

class Plot extends React.Component {
  shouldComponentUpdate(nextProps) {
              const xDataChanged = !this.props.xData.equals(nextProps.xData);
              const yDataChanged = !this.props.yData.equals(nextProps.yData);

              return xDataChanged || yDataChanged;
        }

  drawPlot = () => {
    Plotly.newPlot('plot', [{
      x: this.props.xData.toJS(),
      y: this.props.yData.toJS(),
      type: this.props.type
    }], {
      margin: {
        t: 0, r: 0, l: 30
      },
      xaxis: {
        gridcolor: 'transparent'
      }
    }, {
      displayModeBar: false
    });

    document.getElementById('plot').on(
      'plotly_click', this.props.onPlotClick);
  }

  componentDidMount() {
    this.drawPlot();
  }

  componentDidUpdate() {

    this.drawPlot();
  }

  render() {
    console.log('RENDER PLOT');
    return (
      <div id="plot"></div>
    );
  }
}


export default Plot;
```

Now try clicking around and loading different cities.

The `Plot` component now only rerenders when new data comes in! 🎉

We've entirely gotten rid of the continuous rerenders, and only rerender when it's *really* necessary! This is awesome!

Let's explore how we can make sure our app works the way we expect it to, no matter who's working on it i.e. lets test it.

Additional Material

- Official ImmutableJS docs
- Introduction to Immutable.js and Functional Programming Concepts
- Pros and Cons of using immutability with React.js