Immutable Redux

In this lesson, we will use Immutable.js in our Weather app that's created using React and Redux

we make the initial state in our reducer an immutable object by using the `fromJS` function! We simply wrap the object that we assign to `initialState` in `fromJS` like so:

```
// reducer.js
/* … */
import { fromJS } from 'immutable';

var initialState = fromJS({
  /* … */
});

/* … */
```

Now we need to rework our reducer. Since our state is now immutable, instead of doing `Object.assign({}, state, { /* … */ })` everywhere we can simply use `state.set`!

Let's showcase this on the `CHANGE_LOCATION` action. This is what our reducer looks like right now:

```
case 'CHANGE_LOCATION':
    return Object.assign({}, state, {
      location: action.location
    });
```

Instead of doing this whole assigning business, we can simply `return state.set('location', action.location)`!

```
case 'CHANGE_LOCATION':
    return state.set('location', action.location);
```

Not only is that a lot cleaner, it's also forcing us to work immutably, which

Not only is that a lot cleaner, it's also forcing us to work immutably, which means we can't accidentally mess something up and introduce weird bugs! 🎉

Let's do the same thing for our `SET_DATA`, `SET_DATES` and `SET_TEMPS` cases:

```
case 'SET_DATA':
  return Object.assign({}, state, {
    data: action.data
  });
case 'SET_DATES':
  return Object.assign({}, state, {
    dates: action.dates
  });
case 'SET_TEMPS':
  return Object.assign({}, state, {
    temps: action.temps
  });
```

This whole block becomes:

```
case 'SET_DATA':
 return state.set('data', fromJS(action.data));
case 'SET_DATES':
 return state.set('dates', fromJS(action.dates));
case 'SET_TEMPS':
 return state.set('temps', fromJS(action.temps));
```

Isn't that nice? Now, here's the last trickery in our reducer, because what do we do for `SET_SELECTED_TEMP` and `SET_SELECTED_DATE`? How do we set `state.selected.temp`?

It turns out Immutable provides us with a really nice function for that called `setIn`. We can use `setIn` to set a nested property by passing in an array of keys we want to iterate through! Let's take a look at that for our `SET_SELECTED_DATE`.

This is what it currently looks like:

```
case 'SET_SELECTED_DATE':
  return Object.assign({}, state, {
    selected: {
      date: action.date,
      temp: state.selected.temp
    }
```

```
  });
```

This works, but you have to agree it's not very nice. With `setIn`, we can simply replace this entire call with this short form:

```
case 'SET_SELECTED_DATE':
  return state.setIn(['selected', 'date'], action.date);
```

So beautiful! Let's do the same thing for `SET_SELECTED_TEMP` and we're done here!

```
case 'SET_SELECTED_TEMP':
  return Object.assign({}, state, {
    selected: {
      date: state.selected.date,
      temp: action.temp
    }
  });
```

becomes

```
case 'SET_SELECTED_TEMP':
  return state.setIn(['selected', 'temp'], action.temp);
```

This is what our reducer looks like finally (let's try running our weather app with our new reducer):

```
import { fromJS } from 'immutable';

var initialState = fromJS({
  location: '',
  data: {},
  dates: [],
  temps: [],
  selected: {
    date: '',
    temp: null
  }
});

export default function mainReducer(state = initialState, action) {
  switch (action.type) {
    case 'CHANGE_LOCATION':
      return state.set('location', action.location);
    case 'SET_DATA':
      return state.set('data', fromJS(action.data));
    case 'SET_DATES':
      return state.set('dates', fromJS(action.dates));
```

```
    case 'SET_TEMPS':
      return state.set('temps', fromJS(action.temps));
    case 'SET_SELECTED_DATE':
      return state.setIn(['selected', 'date'], action.date);
    case 'SET_SELECTED_TEMP':
      return state.setIn(['selected', 'temp'], action.temp);
    default:
      return state;
  }
}
```

If you now try to run your app though, nothing will work and you'll get an error like the following:

`bundle.js:100 mapStateToProps() in Connect(App) must return a plain object. Instead received Map { "location": "", "data": Map {}, "dates": List [], "temps": List [], "selected": Map { "date": "", "temp": null } }.`

This is because in our `App` component we have a `mapStateToProps` function that simply returns the entire state! An easy trick would be to return `state.toJS`, kind of like this:

```
function mapStateToProps(state) {
  return state.toJS();
}
```

Let's try this again (*hint: It will work this time*).

```
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);
```

Our example above works but there are two downsides to this approach though:

1. Converting from (`fromJS`) and to (`toJS`) JavaScript objects to immutable data structures is *very performance expensive and slow*. This is fine for the `initialState` because we only ever convert that once, but doing that on every render will have an impact on your app.

2. You thus lose the main benefit of ImmutableJS, which is performance!

Now you might be thinking “But if it’s so expensive, how can ImmutableJS have performance as its main benefit?”. To explain that we have to quickly go over how ImmutableJS works.