Redux-Thunk: A better way to fetch data

Fetching data is Redux apps can be streamlined by using redux-thunk. We are going to use redux-thunk to fetch weather data

WE'LL COVER THE FOLLOWING

The idea behind `redux-thunk` is that we return a function from an action that gets passed `dispatch`. This allows us to do asynchronous things (like data fetching) in our actions:

```
function someAction()
  // Notice how we return a function - this is what's called a "thunk"!
  return function thisIsAThunk(dispatch) {
    // Do something asynchronous in here
  }
}
```

First implementation

Let's try to write an action called `fetchData` that fetches our data! Start with the basic structure:

```
// actions.js

/* …more actions here… */

export function fetchData() {
  return function thunk(dispatch) {
    // LET'S FETCH OUR DATA HERE
  }
}
```

Now let's copy and paste the `xhr` call from the `App` component and put it in there:

```
// actions.js

/* …more actions here… */

export function fetchData() {
  return function thunk(dispatch) {
    xhr({
      url: url
    }, function (err, data) {

      var body = JSON.parse(data.body);
      var list = body.list;
      var dates = [];
      var temps = [];
      for (var i = 0; i < list.length; i++) {
        dates.push(list[i].dt_txt);
        temps.push(list[i].main.temp);
      }

      self.props.dispatch(setData(body));
      self.props.dispatch(setDates(dates));
      self.props.dispatch(setTemps(temps));
      self.props.dispatch(setSelectedDate(''));
      self.props.dispatch(setSelectedTemp(null));
    });
  }
}
```

Now we need to fix three things: 1) We need to import `xhr`, 2) we need to get the URL from the action and 3) we need to rename all `self.props.dispatch` calls to `dispatch`:

```
// actions.js

// REQUIRE xhr
import xhr from 'xhr';

/* …more actions here… */
```

```
// PASS URL IN HERE
export function fetchData(url) {

  return function thunk(dispatch) {
    xhr({
      url: url
    }, function (err, data) {

      var data = JSON.parse(data.body);
      var list = data.list;
      var dates = [];
      var temps = [];
      for (var i = 0; i < list.length; i++) {
        dates.push(list[i].dt_txt);
        temps.push(list[i].main.temp);
      }
      // RENAME self.props.dispatch TO dispatch
      dispatch(setData(data));
      dispatch(setDates(dates));
      dispatch(setTemps(temps));
      dispatch(setSelectedDate(''));
      dispatch(setSelectedTemp(null));
    });
  }
}
```

Well, that was easy! That's our thunked action done – let's call it from our `App` component:

```
/* … */

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

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

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

    this.props.dispatch(fetchData(url));
  },
  onPlotClick = (data) => { /* … */ },
  changeLocation = (evt) => { /* … */ },
```

```
  render() { /* … */ }
});


/* … */
```

That makes our `App` so much nicer to work with already!

Wiring it up

The last step is wiring up `redux-thunk`. `redux-thunk` is a so-called "middleware". Middlewares sit in between the action and the reducers, every action you dispatch gets passed to all middlewares you add. (that's why they're called *middle* ware)!

Now let's `apply` the `thunk` middleware in our `createStore` call in `index.js`:

```
// index.js

/* … */
import { createStore, applyMiddleware } from 'redux';
import thunkMiddleware from 'redux-thunk';

/* … */

var store = createStore(
  mainReducer,
  applyMiddleware(thunkMiddleware)
);
/* … */
```

And that's it, everything should be working again now. Look how easy it is to handle our components, how nicely everything is separeted by concern and how easy it would be to add a new feature to our app! That's the power of redux, our application is easier to reason about and to handle, instead of having one massive top-level `App` component we separate the concerns properly.

Here's our Redux app in action

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

import Plot from './Plot';
```

```
import Plot from 'react-plotly.js';
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;
}

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

And that's it, everything should be working again now. Look how easy it is to handle our components, how nicely everything is separeted by concern and how easy it would be to add a new feature to our app! That's the power of redux, our application is easier to reason about and to handle, instead of having one massive top-level `App` component we separate the concerns properly.

Now, let's find out how we can make our app so much more performant with immutable datastructures

Additional Material

- Official Redux Docs
- André Staltz' Unidirectional User Interface Architectures
- Egghead.io "Getting started with Redux" Video Course
- Egghead.io Advanced Redux Video Course