 



# Step 1: Create your Django project

Ok. So you decided you want to use ReactJS with your new or existing Django project. That's super cool. Since we don't have a project at hand, let's just create a new one:

mkvirtualenv djreact

virtualenv on Windows,

$ cd /your\_path

$ /your\_path > mkdir todo && cd todo

$ /your\_path/todo> virtualenv backend

$ /your\_path/todo> backend/Scripts/activate

(backend) $ /your\_path/todo> pip install django

(backend) $ /your\_path/todo> pip install djangorestframework

(backend) $ /your\_path/todo> cd backend

(backend) $ /your\_path/todo/backend> django-admin startproject todo\_api

(backend) $ /your\_path/todo/backend> cd tod\_api

(backend) $ /your\_path/todo/backend/todo\_api> python manage.py startapp todos

(backend) $ /your\_path/todo/backend/todo\_api> python manage.py migrate

pip install Django

django-admin startproject djreact

mv djreact django

If you have never seen this mkvirtualenv command, you should take some time and learn about [virtualenvwrapper](http://virtualenvwrapper.readthedocs.org/en/latest/) first.

This creates a new Django project in the root folder of your repository. I like to rename that folder to django just because it will sit besides other folders like ansible and maybe even react-native. I like the root folders to describe the main technology that is used within them.

You also want to create a requirements.txt file and put Django==1.9.3 inside. If you have never used a requirements.txt file, have a look at the [pip documentation](https://pip.readthedocs.org/en/1.1/requirements.html).

And finally we should create a .gitignore file and add \*.pyc files and db.sqlite3.

At this point, you can run ./manage.py runserver and you should see the Django welcome page in your browser at http://localhost:8000.

# Step 2: Add non-reactJS views

We want to show that ReactJS can easily be used with an existing project, so we will add a few "legacy-views" to simulate that this is an old existing Django project.

I added the following lines to urls.py:

from django.views import generic

urlpatterns = [

url(r'^admin/', admin.site.urls),

url(r'^view2/',

generic.TemplateView.as\_view(template\_name='view2.html')),

url(r'^$',

generic.TemplateView.as\_view(template\_name='view1.html')),

]

Next, I added a few templates to the templates folder and finally I made sure that Django is aware of these templates by putting this into settings.py:

TEMPLATES = [

{

...

'DIRS': [os.path.join(BASE\_DIR, 'djreact/templates')],

...

},

]

The base-template is the file base.html. It imports the [Twitter Bootstrap CSS Framework](http://getbootstrap.com/):

<!doctype html>

<html class="no-js" lang="">

<head>

<meta charset="utf-8">

<meta http-equiv="x-ua-compatible" content="ie=edge">

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css" integrity="sha384-1q8mTJOASx8j1Au+a5WDVnPi2lkFfwwEAa8hDDdjZlpLegxhjVME1fgjWPGmkzs7" crossorigin="anonymous">

</head>

<body>

{% block main %}{% endblock %}

</body>

</html>

The templates for the two views are view1.html:

{% extends "base.html" %}

{% block main %}

<div class="container">

<h1>View 1</h1>

</div>

{% endblock %}

and view2.html:

{% extends "base.html" %}

{% block main %}

<div class="container">

<h1>View 2</h1>

</div>

{% endblock %}

At this point you can run ./manage.py runserver and you should see "View 1" in your browser. You can change the URL to /view2/ and you should see "View 2".

I'm importing Twitter Bootstrap here because I also want to show that ReactJS will not stand in your way even if you are already using a complex CSS framework. More on this in a later step.

# Step 3: Add django-webpack-loader

Unfortunately, in this step a lot of stuff will happen all at once. This is the step where most people give up, because Webpack is one monster of a tool and super hard to understand and to configure.

Let's try to walk through this step by step.

First of all we need to run pip install django-webpack-loader and of course we will also add it to requirements.txt. Tip: Whenever you install something with pip, run pip freeze immediately after and copy and paste that package with it's version number into your requirements.txt.

Next we need to add this reusable Django app to the INSTALLED\_APPS setting in our settings.py:

INSTALLED\_APPS = [

...

'webpack\_loader',

]

ReactJS is all about creating "bundles" (aka minified JavaScript files). These bundles will be saved in our static folder, just like we always used to do it with our CSS and JS files. So we need to make Django aware of this static folder in settings.py:

STATICFILES\_DIRS = [

os.path.join(BASE\_DIR, 'djreact/static'),

]

Next we need to create a package.json file, which is something similar to Python's requirements.txt file:

{

"name": "djreact",

"version": "0.0.1",

"devDependencies": {

"babel": "^6.5.2",

"babel-core": "^6.6.5",

"babel-eslint": "^5.0.0",

"babel-loader": "^6.2.4",

"babel-plugin-transform-decorators-legacy": "^1.3.4",

"babel-preset-es2015": "^6.6.0",

"babel-preset-react": "^6.5.0",

"babel-preset-stage-0": "^6.5.0",

"eslint": "^2.2.0",

"react": "^0.14.7",

"react-hot-loader": "^1.3.0",

"redux-devtools": "^3.1.1",

"webpack": "^1.12.13",

"webpack-bundle-tracker": "0.0.93",

"webpack-dev-server": "^1.14.1"

},

"dependencies": {

"es6-promise": "^3.1.2",

"isomorphic-fetch": "^2.2.1",

"lodash": "^4.5.1",

"radium": "^0.16.6",

"react-cookie": "^0.4.5",

"react-dom": "^0.14.7",

"react-redux": "^4.4.0",

"redux": "^3.3.1",

"redux-thunk": "^1.0.3"

}

}

I won't explain in detail what each package is good for. Finding out what you really need is essentially one of the really hard parts when starting out with ReactJS. Describing the reasons behind each of these packages would go far beyond the scope of this quick tutorial. A lot of this stuff has to do with [Babel](http://babeljs.io/), which is a tool that "transpiles" cutting edge JavaScript syntax into something that browsers support.

When you have created the file, you can install the packages via npm install **on your Project folder**. This will create a node\_modules folder, so we should also add that folder to .gitignore. If you don't have npm installed, now is a good time to google for it and find out how to install it on your OS.

After you ran **npm install**, you should be able to use **webpack** - in theory.

In praxis, you need to create quite a monstrous config first. I will cheat a little bit and already split it into two files because that will be quite helpful later.

The first file is called **webpack.base.config.js** and looks like this:

var path = require("path")

var webpack = require('webpack')

module.exports = {

context: \_\_dirname,

entry: {

// Add as many entry points as you have container-react-components here

App1: './reactjs/App1',

vendors: ['react'],

},

output: {

path: path.resolve('./djreact/static/bundles/local/'),

filename: "[name]-[hash].js"

},

optimization: {

splitChunks: {

chunks: 'all',

},

},

externals: [

], // add all vendor libs

plugins: [

//new webpack.optimize.CommonsChunkPlugin('vendors', 'vendors.js'),

// it’s deprecated so use optimization in red above.

], // add all common plugins here

module: {

loaders: [] // add all common loaders here

},

resolve: {

modulesDirectories: ['node\_modules', 'bower\_components'],

extensions: ['', '.js', '.jsx']

},

}

It does a lot of things:

1. It defines the entry point. That is the JS-file that should be loaded first
2. It defines the output path. This is where we want to save our bundle.
3. It uses the ~~CommonsChunksPlugin~~, this makes sure that ReactJS will be saved as a different file (vendors.js), so that our actual app-bundle doesn't become too big.

The second file is called **webpack.local.config.js** and looks like this:

var path = require("path")

var webpack = require('webpack')

var BundleTracker = require('webpack-bundle-tracker')

var config = require('./webpack.base.config.js')

config.devtool = "#eval-source-map"

config.plugins = config.plugins.concat([

new BundleTracker({filename: './webpack-stats-local.json'}),

])

config.module.loaders.push(

{ test: /**\.**jsx?$/, exclude: /node\_modules/, loaders: ['react-hot', 'babel'] }

)

module.exports = config

This essentially loads the base config and then adds a few things to it, most notably one more plugin: The BundleTracker plugin.

This plugin creates a JSON file every time we generate bundles. Django can then read that JSON file and will know which bundle belongs to which App-name (this will make more sense later).

We will be using bleeding edge ES2015 JavaScript syntax for all our JavaScript code. A plugin called babel will "transpile" the advanced code back into something that browsers can understand. For this to work, we need to create the following .babelrc file:

{

"presets": ["es2015", "react", "stage-0"],

"plugins": [

["transform-decorators-legacy"],

]

}

Now we could use webpack to create a bundle, but we haven't written any JavaScript or ReactJS code, yet.

First, create a reactjs folder and put a App1.jsx file inside. This is going to be one of our entry-points for bundling. webpackwill look into that file and then follow all it's imports and add them to the bundle, so that in the end we will have one big App1.jsx file that can be used by the browser.

import React from "react"

import { render } from "react-dom"

import App1Container from "./containers/App1Container"

class App1 extends React.Component {

render() {

return (

<App1Container />

)

}

}

render(<App1/>, document.getElementById('App1'))

As you can see, this file tries to import another component called App1Container. So let's create that one as well in the file containers/App1Container.jsx:

import React from "react"

import Headline from "../components/Headline"

export default class App1Container extends React.Component {

render() {

return (

<div className="container">

<div className="row">

<div className="col-sm-12">

<Headline>Sample App!</Headline>

</div>

</div>

</div>

)

}

}

And once again, that component imports another component called Headline. Let's create that one as well in components/Headline.jsx:

import React from "react"

export default class Headline extends React.Component {

render() {

return (

<h1>{ this.props.children }</h1>

)

}

}

You might wonder why I am using a component App1 and another one App1Container. This will make more sense a bit later. We will be using something called Redux to manage our app's state and you will see that Redux requires quite a lot of boilerplate to be wrapped around your app. To keep my files cleaner, I like to have one "boilerplate" file, which then imports the actual ReactJS component that I want to build.

You will also notice that I separate my components into a containers folder and into a components folder. You can think about this a bit like Django views. The main view template is your container. It contains the general structure and markup for your page. In the components we will have much smaller components that do one thing and one thing well. These components will be re-used and orchestrated by all our container components, they would be the equivalent of smaller partial templates that you import in Django using the {% import %} tag.

At this point you can run “node\_modules/.bin/webpack“ --config webpack.local.config.js under your project folder and it should generate some files in **djreact/static/bundles/**.

Like this,

C:\Users\kyung.lee\eclipse-workspv36\DjangoWebProject>"node\_modules/.bin/webpack" --config webpack.local.config.js

Hash: a8f9ac358ab34a0c85d6

Version: webpack 1.15.0

Time: 6980ms

Asset Size Chunks Chunk Names

DjangoWebProject-a8f9ac358ab34a0c85d6.js 1.79 MB 0 [emitted] DjangoWebProject

vendors-a8f9ac358ab34a0c85d6.js 1.78 MB 1 [emitted] vendors

[0] multi vendors 28 bytes {1} [built]

+ 161 hidden modules

# Step 4: Use the bundle

In the last step we have create our first bundle, but we haven't seen the result in the browser. Let's update our template to use our fancy new ReactJS app now.

Change view1.html so that it looks like this:

{% extends "base.html" %}

{% load render\_bundle from webpack\_loader %}

{% block main %}

<div id="App1"></div>

{% render\_bundle 'vendors' %}

{% render\_bundle 'App1' %}

{% endblock %}

We also need to add a new setting to settings.py:

WEBPACK\_LOADER = {

'DEFAULT': {

'BUNDLE\_DIR\_NAME': 'bundles/local/', # end with slash

'STATS\_FILE': os.path.join(BASE\_DIR, 'webpack-stats-local.json'),

}

}

BUNDLE\_DIR\_NAME tells Django in which folder within the static folder it can find our bundle.

STATS\_FILE tells Django where it can find the JSON-file that maps entry-point names to bundle files. It is because of this stats file that we can use {% render\_bundle 'App1' %} in our template. You will also find this App1 name in your webpack.base.config.js file under the entry attribute.

Now run ./manage.py runserver and visit your site. You should see "Sample App!".

Now try to make a change to your ReactJS app. Change Sample App! to Something New! in containers/App1Container.jsx.

Then run node\_modules/.bin/webpack --config webpack.local.config.js again, make sure that ./manage.py runserver is still running and visit your site in the browser. It should say "Something New!" now.

Amazing, huh?

# Step 5: Hot Reloading

Step 4 was nice and awesome, but not mind-blowing. Let's do mind-blowing now. We don't really want to run that webpackcommand every time we change our ReactJS app (and create thousands of local bundles in the process). We want to see the changes in the browser immediately.

First, we need a server.js file that will start a webpack-dev-server for us:

var webpack = require('webpack')

var WebpackDevServer = require('webpack-dev-server')

var config = require('./webpack.local.config')

new WebpackDevServer(webpack(config), {

publicPath: config.output.publicPath,

hot: true,

inline: true,

historyApiFallback: true,

}).listen(3000, config.ip, function (err, result) {

if (err) {

console.log(err)

}

console.log('Listening at ' + config.ip + ':3000')

})

Next, we need to add/replace the following in our webpack.local.config.js:

var ip = 'localhost'

config.entry = {

App1: [

'webpack-dev-server/client?http://' + ip + ':3000',

'webpack/hot/only-dev-server',

'./reactjs/App1',

],

}

config.output.publicPath = 'http://' + ip + ':3000' + '/assets/bundles/'

config.plugins = config.plugins.concat([

new webpack.HotModuleReplacementPlugin(),

new webpack.NoErrorsPlugin(),

new BundleTracker({filename: './webpack-stats-local.json'}),

])

Ready? In one terminal window, start the webpack-dev-server with node server.js and in another terminal window, start the Django devserver with ./manage.py runserver.

Make sure that you can still see "Something New!".

And now change it to Something Fancy! in containers/App1Container.jsx and switch back to your browser. If you are very fast, you can see how it updates itself.

There is another cool thing: When you open the site in Google Chrome and open the developer tools with COMMAND+OPTION+iand then open the Sources tab, you can see webpack:// in the sidebar. It has a folder called . where you will find the original ReactJS sources. You can even put breakpoints here and debug your app like a pro. No more console.log() in your JavaScript code.

# Step 6: Going to production

There are probably a hundred ways to achieve what we want to do in this step. You can take my approach as a suggestion or just apply whatever works best for your case.

One way would be to install all the node dependencies on your server and make sure that during each deployment you generate the bundles and then call collectstatic.

I prefer to keep my servers as simple as possible and generate the bundles locally and commit them to the repository.

Let's assume a typical 2-tier environment where we have a staging server and a production server. Let's assume that both servers have different URLs, so the API endpoints on staging all start with *https://sandbox.example.com/api/v1/* and the ones on production start with *https://example.com/api/v1/* You need to hard-code these values into your bundles, because that's what your user's browsers will download and execute.

We need to create a new **webpack.stage.config.js** file and it looks like this:

var webpack = require('webpack')

var BundleTracker = require('webpack-bundle-tracker')

var config = require('./webpack.base.config.js')

config.output.path = require('path').resolve('./djreact/static/bundles/stage/')

config.plugins = config.plugins.concat([

new BundleTracker({filename: './webpack-stats-stage.json'}),

// removes a lot of debugging code in React

new webpack.DefinePlugin({

'process.env': {

'NODE\_ENV': JSON.stringify('staging'),

'BASE\_API\_URL': JSON.stringify(**'https://sandbox.example.com/api/v1/'**),

}}),

// keeps hashes consistent between compilations

new webpack.optimize.OccurenceOrderPlugin(),

// minifies your code

new webpack.optimize.UglifyJsPlugin({

compressor: {

warnings: false

}

})

])

// Add a loader for JSX files

config.module.loaders.push(

{ test: /**\.**jsx?$/, exclude: /node\_modules/, loader: 'babel' }

)

module.exports = config

And likewise, we need to add **webpack.prod.config.js** and it looks like this:

var webpack = require('webpack')

var BundleTracker = require('webpack-bundle-tracker')

var config = require('./webpack.base.config.js')

config.output.path = require('path').resolve('./djreact/static/bundles/prod/')

config.plugins = config.plugins.concat([

new BundleTracker({filename: './webpack-stats-prod.json'}),

// removes a lot of debugging code in React

new webpack.DefinePlugin({

'process.env': {

'NODE\_ENV': JSON.stringify('production'),

'BASE\_API\_URL': JSON.stringify(**'https://example.com/api/v1/'**),

}}),

// keeps hashes consistent between compilations

new webpack.optimize.OccurenceOrderPlugin(),

// minifies your code

new webpack.optimize.UglifyJsPlugin({

compressor: {

warnings: false

}

})

])

// Add a loader for JSX files

config.module.loaders.push(

{ test: /**\.**jsx?$/, exclude: /node\_modules/, loader: 'babel' }

)

module.exports = config

And because we are now using the DefinePlugin to add environment variables, we should update our webpack.local.config.js like this:

config.plugins = config.plugins.concat([

new webpack.HotModuleReplacementPlugin(),

new webpack.NoErrorsPlugin(),

new BundleTracker({filename: './webpack-stats-local.json'}),

new webpack.DefinePlugin({

'process.env': {

'NODE\_ENV': JSON.stringify('development'),

'BASE\_API\_URL': JSON.stringify('https://'+ ip +':8000/api/v1/'),

}}),

])

We can now create stage and prod bundles like this:

node\_modules/.bin/webpack --config webpack.stage.config.js

node\_modules/.bin/webpack --config webpack.prod.config.js

For us lazy programmers, that's really too much typing, so let's put that into a script. Run **pip install Fabric** and add it to requirements.txt.

Then add the following fabfile.py:

from fabric.api import local

def webpack():

local('rm -rf djreact/static/bundles/stage/\*')

local('rm -rf djreact/static/bundles/prod/\*')

local('webpack --config webpack.stage.config.js --progress --colors')

local('webpack --config webpack.prod.config.js --progress --colors')

Your workflow will now look like this:

1. Start ./manage.py runserver
2. Start node server.js
3. Edit your ReactJS app
4. When done, commit your changes
5. Run fab webpack and commit your new bundles
6. Run a deployment

On your servers, you will need a local\_settings.py where you override the WEBPACK\_LOADER setting like this:

WEBPACK\_LOADER = {

'DEFAULT': {

'BUNDLE\_DIR\_NAME': 'bundles/stage/', # end with slash

'STATS\_FILE': os.path.join(BASE\_DIR, 'webpack-stats-stage.json'),

}

}

And similar for prod, of course, just replace stage with prod.

# Step 7: Add Redux

This step is just a bonus, really. You might want to use some other flux implementation to manage your components' state, but Redux is really nice to work with and the de-facto standard at the moment.

The official [Redux documentation](http://redux.js.org/) is much better than anything that I could ever create, so you should read that. For our purposes, here is what you need to copy & paste:

First you create some **Action Creators** in a new file called reactjs/actions/counterActions.js:

export const INCREASE = "INCREASE"

export function increaseCounter() {

return {type: INCREASE}

}

Any action that can somehow change the state of your app should have a constant and for each constant there should be a function that returns either another function or an object with at least {type: CONSTANT}. Those objects can have more data attached, to them, for example {type: CONSTANT, productId: 1}. This data will be accessible in your reducers.

Speaking of which: Reducers are like stores of data. In Python terms, it's really just a dictionary, usually holding JSON objects that your API has returned. I like to have one reducer-file for each Django model.

Let's create a reducer in a new file called reactjs/reducers/counters.js:

import \* as sampleActions from "../actions/counterActions"

const initialState = {

clicks: 0,

}

export default function counters(state=initialState, action={}) {

switch (action.type) {

case sampleActions.INCREASE:

return {...state, clicks: state.clicks + 1}

default:

return state

}

}

See that {...state, clicks: state.clicks + 1} line there? That's a new JavaScript feature called "destructing". It means: "Create a copy of the object state but replace the attribute clicks with something new". This is the only important thing about reducers: They must **never** mutate the state that is passed into the function. They must always either return the unchanged state or return a new object. This is why most people like to use [immutable.js](https://github.com/facebook/immutable-js), but so far I have been too stupid to use it in my projects with Redux :(

Because we usually have many reducers (i.e. one for every Django model), I like to export them all in a file reactjs/reducers/index.js:

export { default as counters } from './counters'

In one of the earlier steps I mentioned that Redux requires to setup quite a bit of boilerplate around your root-component. This is what we will do now in App1.jsx:

import React from "react"

import { render } from "react-dom"

import {

createStore,

compose,

applyMiddleware,

combineReducers,

} from "redux"

import { Provider } from "react-redux"

import thunk from "redux-thunk"

import \* as reducers from "./reducers"

import App1Container from "./containers/App1Container"

let finalCreateStore = compose(

applyMiddleware(thunk),

window.devToolsExtension ? window.devToolsExtension() : f => f

)(createStore)

let reducer = combineReducers(reducers)

let store = finalCreateStore(reducer)

class App1 extends React.Component {

render() {

return (

<Provider store={store}>

<App1Container />

</Provider>

)

}

}

render(<App1/>, document.getElementById('App1'))

That's a lot of magic that will make more sense to you when you read the full Redux documentation. Basically we are importing all our reducers and composing them into a store and then we wrap that <Provider /> component around our actual root-component.

Next we can upgrade our App1Container to make use of Redux via the @connect decorator:

import React from "react"

import { connect } from "react-redux"

import \* as counterActions from "../actions/counterActions"

import Headline from "../components/Headline"

@connect(state => ({

counters: state.counters,

}))

export default class SampleAppContainer extends React.Component {

handleClick() {

let {dispatch} = this.props;

dispatch(counterActions.increaseCounter())

}

render() {

let {counters} = this.props

return (

<div className="container">

<div className="row">

<div className="col-sm-12">

<Headline>Sample App!</Headline>

<div onClick={() => this.handleClick()}>INCREASE</div>

<p>{counters.clicks}</p>

<p>{process.env.BASE\_API\_URL}</p>

</div>

</div>

</div>

)

}

}

Do you see now why I like to have App1.jsx as an entry point and App1Container.jsx as the actual root-component? As we "reactify" parts of our existing Django app step by step, each of our ReactJS apps will want to have access to Redux, so we will put that boilerplate around it's entry file. However at some time in the future we might reach a point where our ReactJS codebase is larger than our Django-template codebase and we might want to make the last final step and migrate everything over to 100% React. We will end up with just one single entry point (if we can turn the site into a SPA) or at least much lesser entry-points than before, so we can just delete those files in the reactjs root folder. We would then probably use something like react-router to compose our actual root-components in the containers folder.

Oh and by the way! You should totally install this Chrome Extension: <https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd>

Once you have that and you visit your site, you can open the developer tools with COMMAND+OPTION+i and there will be a Reduxtab which will show you all actions that are being fired and the new values in your reducers after the action has fired. This is unbelievably helpful for debugging! Try to run node server.js and ./manage.py runserver and click at the "INCREASE" link.

# Step 8: Inline Styles

This, to me, is the greatest advancement in web development since the invention of CSS. Do you have humongous stylesheets in your projects where no one ever dares to delete a style because nobody knows where in the project it might still be in use? React solves that.

Do you use LESS or SASS and build nicely nested styles like this:

.some-container {

.some-inner-container {

.active {

...

}

}

}

And then you want to re-use that .active style on some new element but you can't because that element would have to be wrapped in those other elements that have the outer styles. React solves that as well.

Update your App1Container.jsx to use Radium for styles:

import React from "react"

import Radium from "radium"

import { connect } from "react-redux"

import \* as counterActions from "../actions/counterActions"

import Headline from "../components/Headline"

const styles = {

button: {

cursor: "pointer",

},

counter: {

color: "blue",

fontSize: "20px",

}

}

@connect(state => ({

counters: state.counters,

}))

@Radium

export default class App1Container extends React.Component {

handleClick() {

let {dispatch} = this.props;

dispatch(counterActions.increaseCounter())

}

render() {

let {counters} = this.props

return (

<div className="container">

<div className="row">

<div className="col-sm-12">

<Headline>Sample App!</Headline>

<div style={[styles.button]} onClick={() => this.handleClick()}>INCREASE</div>

<p style={[styles.counter]}>{counters.clicks}</p>

<p>{process.env.BASE\_API\_URL}</p>

</div>

</div>

</div>

)

}

}

That's it! You define a styles object, wrap your class with the @Radium decorator and then use your styles. Remember: It's all just JavaScript. You will quickly want a theme.js file somewhere in your project that all your components can import so that you can re-use commonly used values like font-family, font sizes, colors etc.

So far, I have never put anything other than constants into my theme.js. All other markup and component-specific styling happens in the component itself. Now it is save to change, add and delete styles right there in the component, because it will only affect that component and developers can see at one glance what styles are there and where they are used.

# Step 9: Fetching Data

This is another thing where you have 100 different options and I'm not sure if this is the best way to do it. There are new technologies evolving like Relay and GraphQL that will make this step even easier. For me, the technique presented here works very well for a mobile app and ReactJS web-components in production, so I guess it's OK to share this.

At the moment, our components are pretty dumb because they can't fetch data from the outside world (i.e. API endpoints). We are going to use [fetch](https://github.com/github/fetch) which is a Polyfill for Browsers and does something similar to jQuery.ajax().

In this example, let's try to fetch all my repos from the public Github API and display them.

The problem with HTTP requests is that so many things can go wrong. The server might return a 404 or a 500 error or might be completely offline so that the request times out or you might even not have a network connection at all. In order to deal with all these errors, I wrote a little wrapper around the fetch function. Let's create a file utils.js in the reactjs folder:

import fetch from "isomorphic-fetch"

export function request(url, options, success, error400, error, failure) {

let headers = new Headers()

headers.append("Content-Type", "application/json")

headers.append("Accept", "application/json")

options["headers"] = headers

return fetch(url, options)

.then(res => {

if (res.status >= 200 && res.status < 300) {

// for anything in 200-299 we expect our API to return a JSON response

res.json().then(json => { return success(json) })

} else if (res.status === 400) {

// even for 400 we expect a JSON response with form errors

res.json().then(json => { return error400(json) })

} else {

// For all other errors we are not sure if the response is JSON,

// so we just want to display a generic error modal

return error(res)

}

}).catch((ex) => { return failure(ex) })

}

Let's break this down. You will notice that we are actually not using Github's fetch but something else called isomorphic-fetch. This makes sure that your fetch() calls work both on the server (with Node.js) and in the browser.

The wrapper function that I "invented" is called request. It takes a lot of parameters:

1. The url that should be called
2. An object called options that allows us to pass further options into the fetch function - we are not using this in this example.
3. A callback function called success - This function will be executed when fetch returns a successful response. Before calling the function, we will parse the response through .json().
4. A callback function called error400 which will be called if fetch returns with a 400 error.
5. A callback function called error which will be called if fetch returns with any other kind of error, for example a 500 Internal Server Error.
6. A callback function called failure which will be called if the fetch call completely crashed, for example because you don't have a network connection.

This looks all really scary, but this is one of those files that you write once and then never look at it again. So if you want, you can just copy and paste this and then forget about it.

Now we need to create action creators for Redux. Create a file actions/githubActions.js with the following content:

import { request } from "../utils"

export const FETCH\_REPOS = "FETCH\_REPOS"

export const FETCH\_REPOS\_SUCCESS = "FETCH\_REPOS\_SUCCESS"

export const FETCH\_REPOS\_ERROR400 = "FETCH\_REPOS\_ERROR400"

export const FETCH\_REPOS\_ERROR500 = "FETCH\_REPOS\_ERROR500"

export const FETCH\_REPOS\_FAILURE = "FETCH\_REPOS\_FAILURE"

export function fetchRepos() {

return function (dispatch) {

let url = "https://api.github.com/users/mbrochh/repos"

dispatch({type: FETCH\_REPOS})

return request(

url, {},

(json) => { dispatch({type: FETCH\_REPOS\_SUCCESS, res: json}) },

(json) => { dispatch({type: FETCH\_REPOS\_ERROR400, res: json}) },

(res) => { dispatch({type: FETCH\_REPOS\_ERROR500, res: res}) },

(ex) => { dispatch({type: FETCH\_REPOS\_FAILURE, error: ex}) },

)

}

}

As you can see, we are importing our very own request function which we have just created. Then we create constants for the different kinds of events that this action can trigger. Those are the very same events that can happen in a typical fetch call, namely "success", "error 400", "any other error" and "network failure".

Next we write a function called fetchRepos(). Now something weird is going on. From the action creator in counterActions.js you might remember that the action creator is supposed to return an object that looks like this:

return {type: SOME\_CONSTANT}

This works fine for simple actions that just do one thing. In this case our action can trigger many different events and to make things worse, they happen asynchronously (we don't know when the fetch request will return). For this use-case, Redux allows us to write an action creator that doesn't return an object like above but returns another function that takes the dispatchparameter. Inside of this function we can do whatever we want and dispatch actions directly.

The first thing that we do is to dispatch "FETCH\_REPOS", this signals that we have started fetching Github repositories. Then we call our request function and pass in the url, an empty object for options and four anonymous functions as callback functions for the various fetch events. Each callback function does nothing else than return a Redux compatible object with the corresponding event and the event data (i.e. the JSON response or the error object).

Now that we have action creators, we can implement a reducer that listens to those actions. Create a file reducers/github.js:

import \* as githubActions from "../actions/githubActions"

const initialState = {

isLoadingRepos: false,

repos: undefined,

}

export default function github(state=initialState, action={}) {

switch (action.type) {

case githubActions.FETCH\_REPOS:

return {...state, isLoadingRepos: true}

case githubActions.FETCH\_REPOS\_SUCCESS:

return {...state, isLoadingRepos: false, repos: action.res}

case githubActions.FETCH\_REPOS\_ERROR400:

case githubActions.FETCH\_REPOS\_ERROR500:

case githubActions.FETCH\_REPOS\_FAILURE:

return {...state, isLoadingRepos: false}

default:

return state

}

}

Reducers are always the same: You have some initial state where everything is empty and then you have a function with a big switch-statement. In the switch-statement you define which events you are listening to - you can see that those are basically just the constants that we defined in our action creator file. Every time this reducer picks up an action that it is interested in, it will return a copy of state and change one or more fields of the state to a new value.

You also need to update your reducers/index.js so that this new reducer is added to the list of reducers that your app is aware of:

export { default as counters } from "./counters"

export { default as github } from "./github"

Finally you can now use your new reducer in your components. For this, we need to change a lot of code in containers/App1Container.jsx.

First of all, import your new action creator. I'm also importing a new component called GithubRepos here which we will implement next a bit further down.

import \* as counterActions from "../actions/counterActions"

import \* as githubActions from "../actions/githubActions"

import Headline from "../components/Headline"

import GithubRepos from "../components/GithubRepos"

Next you need to connect the new reducer to your component. Again, the key (i.e. counters or github) defines how the reducer is named in this component's props (so it will be this.props.github) and the value (i.e. state.github) refers to the name with which you imported the reducer in reducers/index.js:

@connect(state => ({

counters: state.counters,

github: state.github,

}))

Now you need to make sure that your component calls the new fetchRepos action as soon as it mounts:

componentDidMount() {

let {dispatch, github} = this.props

if (!github.isLoadingRepos && github.repos === undefined) {

dispatch(githubActions.fetchRepos())

}

}

I like to add a renderLoading() function to my components that displays a loading screen while it is fetching it's data:

renderLoading() {

return (

<div className="container">

<div className="row">

<div className="col-sm-12">

Loading...

</div>

</div>

</div>

)

}

And finally we update our render() function to either show the loading screen or show the full app:

render() {

let {counters, github} = this.props

if (github.isLoadingRepos || github.repos === undefined) {

return this.renderLoading()

}

return (

<div className="container">

<div className="row">

<div className="col-sm-12">

<Headline>Sample App!</Headline>

<div style={[styles.button]} onClick={() => this.handleClick()}>INCREASE</div>

<p style={[styles.counter]}>{counters.clicks}</p>

<p>{process.env.BASE\_API\_URL}</p>

{github.repos !== undefined &&

<GithubRepos repos={github.repos} />

}

</div>

</div>

</div>

)

}

You can see that in the case where {github.repos !== undefined && } we want to render our new GithubRepos component, so we need to implement this component as well. Create a new file components/GithubRepos.jsx:

import React from "react"

export default class GithubRepos extends React.Component {

render() {

let {repos} = this.props

let repoNodes = []

repos.forEach((item, index) => {

let node = (

<div key={item.id}>{item.name}</div>

)

repoNodes.push(node)

})

return (

<div>{repoNodes}</div>

)

}

}

All it does is iterating over the repos that have been passed in as props and then creating a new node for each component. NOTE: When you create components in a loop like this, you must give them a unique key attribute.

All this is quite a lot to take in. It took me several months to come up with this workflow but once you have your utils.js in place, the worst suffering is over and you can just focus on writing action creators and reducers, which is actually quite fun!