Web Development: Module 2, Lesson 7  
Developing React front-end Hands-On Lab

## Overview

Building on the [Module 2 Lesson 6 Lab](https://github.com/MSFTImagine/computerscience/tree/master/Complimentary%20Course%20Content/Module2/Labs), we will implement a front-end UI so users can see microposts on our microblog, add new posts and remove old posts - CRUD except for no update but you can delete and create a new one with old content.

## Objectives

In this hands-on lab you will learn how to:

* Implement React front-end application which displays blog posts and allows for creation of new posts and deletion of old posts

## Prerequisites

The following are required to complete this hands-on lab:

* A code editor
* Windows PowerShell, Mac Terminal, or some other shell with node.js and npm installed
* You should have completed [Module 2 Lessons 1, 2, 3, 4, and 5](https://github.com/MSFTImagine/computerscience/tree/master/Complimentary%20Course%20Content/Module2/Lessons) as well as the [corresponding labs](https://github.com/MSFTImagine/computerscience/tree/master/Complimentary%20Course%20Content/Module2/Labs).

## Exercises

This hands-on lab includes the following exercises:

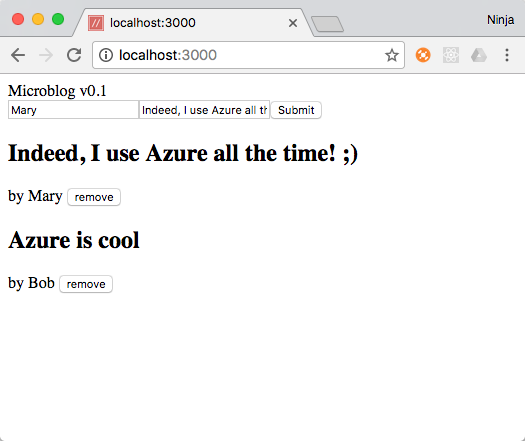
* Exercise 1: Developing React Front-end

## Exercise 1: Implementing a React Front-end

In this exercise, you will implement the front-end UI for a small blog using React. Note: Be sure to refer to the [Module 2 Lessons](https://github.com/MSFTImagine/computerscience/tree/master/Complimentary%20Course%20Content/Module2/Lessons) throughout this exercise.

1. Use [code/lesson7/Lab/package.json](https://github.com/MSFTImagine/computerscience/tree/master/Complimentary%20Course%20Content/Module2/code/lesson7/lab) to configure babel
2. Use [code/lesson7/Lab/package.json](https://github.com/MSFTImagine/computerscience/tree/master/Complimentary%20Course%20Content/Module2/code/lesson7/lab) and npm i to install: react, react-dom, babel-cli, babel-preset-es2015, and babel-preset-react
3. Implement React app (components/app.jsx) which fetches existing posts from the RESTful API using fetch, and shows them in a list
4. Implement in your React app functionality to send POST requests to create a new blog post
5. Implement in your React app functionality to send DELETE requests to remove blog posts
6. Use npm run build or npm run build-watch to compile JSX
7. Compare your solution with [code/lesson7/Lab/components/app.js](https://github.com/MSFTImagine/computerscience/tree/master/Complimentary%20Course%20Content/Module2/code/lesson7/lab/components)x.

The end result of the application is shown in the picture below:



We will start the implementation with creating a new folder as we often do:

mkdir microblog

cd microblog

The project structure will look like this when we are done:

module-2-lesson-7-lab

/components

app.js

/node\_modules

/public

/js

bundle.js

fetch.polyfill.js

index.html

app.js

app.test.js

package.json

README.md

start.sh

test.sh

Right after that, copy the files from the previous lab in which we created a REST API which uses Azure Table. We will build our front-end React interface on top of the RESTful API code. However, we will need to use additional dependencies such as these:

* Babel: We need this tool for converting JSX (a special language for React) into regular JavaScript which will run in the browser.
* React: A front-end library for web User Interface

The package.json file you copies from the previous lab does NOT have all the dependencies we need. You have an option of copying package.json from the final source code for this lab and running npm i, or installing the packages manually using this command:

npm i babel-cli@6.18.0 babel-preset-es2015@6.9.0 babel-preset-react@6.5.0 babel-core@6.18.2 -D

If you going the manual route, then in addition to the dev dependencies, we also need to add a few configurations for the Babel. You can do it right there in the package.json. Add the following lines which start with babel after devDependencies:

"devDependencies": {

"babel-cli": "^6.9.0",

"babel-preset-es2015": "^6.9.0",

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

"expect.js": "0.3.1",

"mocha": "2.5.3",

"superagent": "0.20.0"

},

"babel": {

"presets": [

"es2015",

"react"

]

}

One last step, in regards to the package.json file, is the npm scripts for testing, compilation and development. I recommend creating two more scripts in addition to test and start which you already have from the previous project. The new scripts are build and build-watch. The former will compile JSX into regular JavaScript which we need to do in order to run our React code in the browser. The latter will also compile the code, but it will stay running to monitor for any file changes which is great for development because this way developers won't have to re-run the task over and over. The new scripts use Babel command with --out-file and --watch options. The source maps is a useful feature which will allow to see the JSX line numbers and code in DevTools and not the compiled code line numbers.

"scripts": {

"start": "sh start.sh",

"test": "sh test.sh",

"build": "./node\_modules/.bin/babel components/app.jsx --out-file public/js/bundle.js --source-maps inline",

"build-watch": "./node\_modules/.bin/babel components/app.jsx --watch --out-file public/js/bundle.js --source-maps inline"

},

Make sure it's a valid JSON, i.e., you have all the commas and all the double quotes in the right place. If you experience any type of issues while manually enhancing the package.json file, refer to the final version in the source code for this lab.

**Implementing React Component**

The source code for the browser script will be in components/app.jsx. At the high level, it'll look like this:

"use strict";

let baseUrl = '/api'

let fD = ReactDOM.findDOMNode

let App = React.createClass({

getInitialState() {

return {posts: null} // Set the initial value to null

},

loadPosts() {

// Make a GET /posts request to the REST API to fetch existing posts

},

componentDidMount() {

this.loadPosts() // Call method to fetch posts from the back-end

},

render() {

// Render AddPost and PostList components with props

}

})

let AddPost = React.createClass({

handleSubmit(event) {

// Make a POST /posts request to the REST API to save a new post

},

render() {

// Render new post form with inputs

}

})

let PostList = React.createClass({

render() {

// Render list of posts (if any)

}

})

let Post = React.createClass({

removePost() {

// Make a DELETE request to REST API

},

render() {

// Render individual post view with a remove button

}

})

ReactDOM.render(<App/>, document.getElementById('app')) // Mount the main component

If the structure above is understood, then let's proceed to the implementation of the methods in the three components. For AJAX/XHR requests to the server, we will be using fetch API. If you have to support an old browser which does not support fetch API, then in the index.html, we will include a polyfill to provide the fetch functionality.

First of all, we set the mode to strict which will tell browsers that we are using ES6/ES2015 syntax. Then, we store the base URL to the REST API. In our case, it's hosted on the same domain so there's no need for the domain name. Also, we'll be using ReactDOM.findDOMNode quite a lot so it pays to save it in a short variable fD.

"use strict";

let baseUrl = '/api'

let fD = ReactDOM.findDOMNode

Right after that we create our main component App which will have the logic to fetch the list of posts. It's absolutely necessary to initialize the state by setting it to some value like null. The fetching is triggered in componentDidMount() lifecycle event which is (as you can guess from its name) is executed when this particular component is mounted to the DOM of the page. In the render() method, we return two other components AddPost and PostList which we still need to implement. We pass the loadPosts() method as a property to AddPost so it can trigger the fetching of the posts upon addition of a new post. We do the same for the PostList. We also pass the list of posts to PostList, because that's what PostList is supposed to do—render a list of posts.

let App = React.createClass({

getInitialState() {

return {posts: null}

},

loadPosts() {

fetch(baseUrl + '/posts')

.then((response) => {

return response.json() // The JSON payload is returned by the response.json() method

}).then((body) => {

this.setState({posts: body}) // Update the state and thus update the view

})

},

componentDidMount() {

this.loadPosts()

},

render() {

return (

<div>Microblog v0.1

<AddPost loadPosts={this.loadPosts}/>

<PostList posts={this.state.posts} loadPosts={this.loadPosts}></PostList>

</div>

)

}

})

Next, we can implement AddPost which will have an event handler and the code to make a POST request as well as the form with inputs in its render().

let AddPost = React.createClass({

handleSubmit(event) {

event.preventDefault()

fetch(baseUrl + '/posts', {

method: 'POST',

headers: { // Set headers to ensure that the server interprets the content properly

'Accept': 'application/json',

'Content-Type': 'application/json'

},

body: JSON.stringify({ // Convert object to a string

author: fD(this.refs.author).value,

text: fD(this.refs.text).value,

})

}).then((response)=>{

this.props.loadPosts() // On response, trigger fetching of the list of posts

})

},

render() {

return (

<form onSubmit={this.handleSubmit}>

<input name="author" type="text" ref="author" placeholder="Peter"/>

<input name="text" type="text" ref="text" placeholder="I'm learning Node.js!"/ >

<input type="submit"/>

</form>

)

}

})

Immediately after AddPost, we define PostList component. It's rather simple as it has only one method render() in which it uses conditions to either show *Loading...* (when GET /posts request is being made), or *No posts yet* (when response has no posts). When rendering the list itself, we are using map() with the key attribute for the Post component. Key will help React to find a particular row faster. RowKey from Azure Table is what we can use for unique value. The main post data such as author and text is not rendered here, but in Post.

let PostList = React.createClass({

render() {

if (this.props.posts == null) return <div>Loading...</div>

if (this.props.posts.length == 0) return <div>No posts yet</div>

return (

<div>

{this.props.posts.map((post)=>{

return <Post key={post.RowKey.\_} post={post} loadPosts={this.props.loadPosts}/>

})}

</div>

)

}

})

Finally, we need to implement one last component. It's an individual post component which will have a DELETE call to the API. In the render() function, it displays the post info such as author and text using Azure Table structure author.\_ and text.\_. The removal method is triggered by a click on the button due to the event handler onClick which we define in the <button>.

let Post = React.createClass({

removePost() {

fetch(`${baseUrl}/posts/${this.props.post.RowKey.\_}`, {

method: 'DELETE',

headers: {

'Accept': 'application/json',

'Content-Type': 'application/json'

}}).then((response)=>{

console.log(response)

this.props.loadPosts()

})

},

render() {

let post = this.props.post

return <div><h2>{post.text.\_}</h2> by {post.author.\_} <button onClick={this.removePost}>remove</button></div>

}

})

ReactDOM.render(<App/>, document.getElementById('app'))

One more thing that we need to do before trying to run this app is to create an HTML file. Open Explorer or Finder or do it in your terminal / command prompt window. We need to create a folder named ‘public’ and then a folder ‘js’ as a subdirectory of ‘public.’ Put fetch.js into public/js. Now, create index.html in public.

It will have a few script tags for React, React DOM and fetch polyfill. In the body, make sure to create a <div> with ID app and a script tag which includes js/bundle.js. We don't create js/bundle.js manually. Babel will do this for us when we run npm run build. As for React, ReactDOM and fetch polyfill (only for old browsers), you can download them on the web: [React](https://unpkg.com/react@15/dist/react.js), [ReactDOM](https://unpkg.com/react-dom@15/dist/react-dom.js) and [Fetch polyfill](https://github.com/github/fetch/blob/master/fetch.js).

<!DOCTYPE html>

<html>

<head>

<script src="js/react.js"></script>

<script src="js/react-dom.js"></script>

<script src="js/fetch.js"></script>

</head>

<body>

<div id="app"/>

<script src="js/bundle.js"></script>

</body>

</html>

At this point, we are almost ready to run the app. We just need to tell our Express server to act as a web server for HTML, JS and CSS files, not just as a REST API. To do so, we are going to apply static middleware: app.use(express.static('public')). In this statement, public is the folder which has our static files. Any file requested by a browser which is in that folder will be server to the browser. We don't need to do anything manually for index.html or for bundle.js since they are already in (or will be) in public. Let's apply this new middleware right after we create instance of Express server inside app.js:

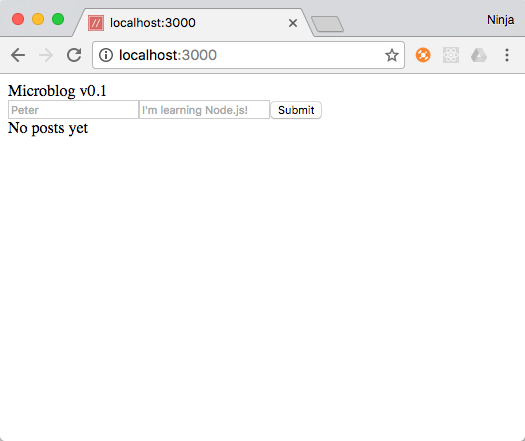
var app = express()

app.use(express.static('public'))

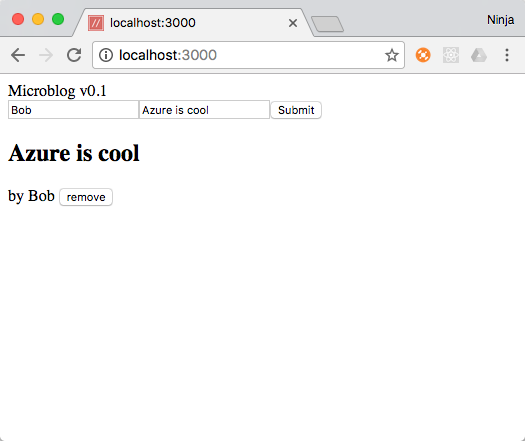
app.use(bodyParser.json())

At last, go to the terminal and run npm run build or npm run build-watch (recommended). You should see a newly created file js/bundle.js. If not, check for any errors in syntax.

Now, open a new terminal window and launch the server with npm start. Go to http://localhost:3000 and observe the app as shown in the screenshot below.



Adding a new post will save it in the database in the cloud and show it on the screen as shown below.



Needless to say, clicking on remove should get rid of a particular post in this microblog app.

Good job. You build a full stack JavaScript app using Node, Express, React and Azure Tables as a persistent NoSQL cloud data storage.

## Summary

In this hands-on lab, you learned how to:

* Use React.js to implement the front-end for a simple blog
* Make AJAX/XHR requests to the REST API server
* Use Babel to transpile JSX to regular JavaScript