# Part 6: Build A React.js Frontend for Your PayPal Clone

*Author:* [*David P. Lopez*](http://www.DavidPLopez.com)

*Publisher:* [*UniqueSoftwareDevelopment*](https://www.uniquesoftwaredev.com)

This is a continuation of our multi-part series on building a simple web application on AWS using AWS Lambda and the ServerlessFramework. You can review the first and second parts of this series starting with the setup of your local environment at:

* [Part 1: How To SetUp Your local Serverless Environment](https://github.com/lopezdp/TechnicalArticles/blob/master/HowToSetUpYourLocalServerlessEnvironment.md)
* [Part 2: How To Configure Your Serverless Backend API](https://github.com/lopezdp/TechnicalArticles/blob/master/HowToConfigureYourServerlessBackend.md)
* [Part 3: How To Configure Your Infrastructure As Code, Mock Services, & Unit Testing](https://github.com/lopezdp/TechnicalArticles/blob/master/HowToConfigure.IAC.Mocks.UnitTests.md)
* [Part 4: How To Deploy and Configure an effective CI/CD Pipeline on AWS](https://github.com/lopezdp/TechnicalArticles/blob/master/HowToReviewServiceToConfigureCICDpipeline.md#part-4--code-review-deploy--configure-an-effective-cicd-pipeline-on-aws)
* [Part 5: How To Implement and Deploy your own Serverless + MicroService to AWS](https://github.com/lopezdp/TechnicalArticles/blob/master/HowToBuildAServerlessMicroService.md)

You can also clone a sample of the application that we will be using in this tutorial at: [PayMyInvoice B2B ClientWallet](https://github.com/lopezdp/pay-me-app)

Please refer to the repo above as you follow along with this tutorial. In this part of the series we will cover implementation steps for a React.js *Single Page Application* that we will deploy on AWS using the following React.js and AWS *imports* and *SDK’s*:

* react
* react-dom
* react-router
* react-router-dom
* react-bootstrap
* aws-amplify

This frontend client discussed above uses the backend logic provided by the implementation of the [PayMyInvoice B2B Wallet](https://github.com/lopezdp/invoice-log-api) deployed to the AWS Cloud as described in the previous chaptes in this series.

## Choose your Framework

Forgive me. I have decided on our frontend framework for you. You are lucky. Twenty (20) years ago the framework thrown on your lap may very well have been ColdFusion, an interesting language to say the least. I purposely choose to not call it a dead language because the fact is they are all very much alive and in *production*, powering many applications that you probably give little thought to. I bring this up because I would like to share a few ideas before fully diving into working together using the hottest framework on the market.

1. Learning how to read, understand, and debug someone else’s code and logic, is more important than understanding how to implement a new React.js application from scratch. You are more than likely going to inherit someone else’s buggy application before being assigned the task to implement a React.js application from scratch. To be able to debug the project in front of you to deploy it for the people who hired you, is the most valuable skillset you can learn right now. Take a look at the comments I found when reviewing the code I was given the responsibility of deploying at one of the first projects I was ever assigned as a professional software engineer:

**Buggy Application Functions**

![alt text](data:text/html; charset=utf-8;base64,)

alt text

1. When implementing a React.js application you must remember to never call setState from within the render() function or your new React Component will produce nothing but a series of infinite loops. Your render function should be a *functionally pure function* that you will use to change the view displayed to the user depending on the state of the application.

Now that we have a bunch of fancy backend logic on our AWS Cloud, we need to develop an engaging user interface that we can use to attract real *humans* to our latest *get rich quick idea*, we are going to continue to call it a digital *Wallet* to make mom, and our resumes, proud. Rather than get into the long history that led to React.js’ role as the ‘V’ or the *View* in the *Model-View-Controller* design paradigm, we will spend some time implementing React.js so that you can learn by doing.

Since the Facebook Mafia is much smarter than us, we are just going to go ahead and use something called [Create React App, or create-react-app for those who are more programatically inclined](https://github.com/facebook/create-react-app). All sarcasm and Dilber Humor aside, React.js is a practical approach to another one of those esoteric software engineering questions that exist in an abundance of threads on the internet that scroll of into perpetuity. Implementing frontend component in React.js is effective and more productive approach that allows you to reuse code that you have written throughout your code. React takes advantage of a Virtual DOM that ituses to check against state changes to render dynamic UI elements faster than traditional MVC implementations.

### Start and Create a New React.js Project

We will have to navigate out of the *serverless + microservice* project directory where we implemented our backend microservices on serverless, and we will have to create a new project we can use for our React.js application. From your home directory, or wherever it is that you keep your project files, go ahead and create a new project by running the following command:

$ npx create-react-app pay-me-app

Make sure that you use the same naming convention that I have used above because create-react-app will not accept capital letters or camelCase conventions. The image below is the output you should receive from the terminal after the installation of create-react-app for the pay-me-app that we are going to develop in this part of the tutorial series.

**React.js App Success!**

![alt text](data:text/html; charset=utf-8;base64,)

alt text

Taking it a bit further, we have to verify that React.js did install iteslf correctly onto our local machine. It’s nice that the terminal says everything is fine, but it’s always good to confirm right?

“Trust but Verify.” - Ronald Reagan

Go ahead and navigate into the new project directory with $ cd ~/pay-me-app and run the following command:

$ npm start

This should *spin up* the local resources that you need to launch your new React.js application in your *browser* and you should be able to see the following generic React.js application.

**React.js App Launched in Browser!**

![alt text](data:text/html; charset=utf-8;base64,)

alt text

### Small Changes

If this is your first React.js application, we can understand why you may be so excited about the output on your browser right now. But where do you start? Let’s keep it simple, and change the title in our application’s index.html. Navigate to:

* $ cd ~/pay-me-app/public

Open the index.html file in the project directory and edit the title tag so that we can name our application accordingly. Below is how I named my version of the application locally:

<title>Pay Me - An online general ledger to track your income.</title>

You can see the changes made at the repository we have created for you on GitHub for the [pay-me-app](https://github.com/lopezdp/pay-me-app).

Facebook did a wonderful job with the [Create React App] framework to make it easy for us to quickly get started with any frontend project. They packaged an easy to use, and compact development library that includes tools that we can use to enable testing, refreshing our app when our files change, by restarting our app in the background, and it also includes ES6 support.

### UIKit with React-Bootstrap

The most important part of any web development project is making sure to have a good set of tools that will let you create a beautiful user interface that your user can use to interact with your serverless backend. You can definitely spend your days implementing your own elements in JavaScript, HTML, and CSS and maybe create your own **UIKit** that you can market to the world to become famous, but for the purposes of this tutorial we are just going to go ahead and use the [Bootstrap UI toolset via the React-Bootstrap library](https://react-bootstrap.netlify.com) that we will import into each of our components as needed. We will install it and make it project dependency now with the following command:

$ npm install react-bootstrap bootstrap

This will make sure to add react-bootstrap as a project dependency in our package.json file that we have saved in our [pay-me-app](https://github.com/lopezdp/pay-me-app) project directory.

Your output will look similar to the following:

+ react-bootstrap@1.0.0-beta.9  
added 30 packages in 11.046s

#### Mobile First Styling

Next we need to configure and add the styling that Bootstrap uses and we will have to add some code to our index.html file to be sure that we take advantage of the standard Bootstrap v4 styles that we will implement here. In your public.html file add the following line:

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css" integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T" crossorigin="anonymous" />

Furthermore, to prevent our user’s screen from zooming in automatically when they focus on an area of their device, we need to change the font size to 16px so that it is large enough to trick the device into not zooming in on focus. We need to make the changes in the file that you can find in your project at: src/index.css where you will need to add the following:

select.form-control,  
textarea.form-control,  
input.form-control {  
 font-size: 16px;  
}  
  
input[type=file] {  
 width: 100%;  
}

In the name of serving as many users as possible, we need to make sure that our application can be viewed elegantly from any mobile device above all else. With the styling above, we are trying to make sure that none of the design elements overflow off the screen to force the browser to add a scroll bar. We have to set the inputs file’s width above to let React.js take all screen sizes and browsers into consideration to normalize our changes across all devices for us.

### React Router for Route Handling Skills

We now need to consider a way to route the different requests that we make available to the users of our Single Page Application. We will take advantage of the nature of how components work within React.js and we will use something called [React Router](https://reacttraining.com/react-router/web/guides/quick-start) to use its collection of *navigational components* that will let us easily declare the routes that our applications needs to handle for our users and their requests.

Lets install it first and make sure it is saved as a dependency in our package.json:

$ npm install react-router-dom

#### Configure React Router

We need to refactor the default files provided to us when we installed the create-react-app framework and we need to configure our new route-handling library so that we can encapsulate our application as a component within React.js so that our application can properly handle our user’s requests. We need to change the App component in our project so that it looks like this:

import React from 'react';  
import ReactDOM from 'react-dom';  
import './index.css';  
import App from "./App";  
import { BrowserRouter as Router } from "react-router-dom";  
import \* as serviceWorker from "./serviceWorker";  
  
ReactDOM.render(  
 <Router>  
 <App />  
 </Router>,  
 document.getElementById('root')  
);  
  
// If you want your app to work offline and load faster, you can change  
// unregister() to register() below. Note this comes with some pitfalls.  
// Learn more about service workers: https://bit.ly/CRA-PWA  
serviceWorker.unregister();

The example above is what enables us to implement and handle the routes that we need to use with our <Router> to render the <App /> component that was initially provided to us by create-react-app. We also want to use the [History](https://developer.mozilla.org/en-US/docs/Web/API/History) interface in JavaScript that will allow us to manipulate our browser’s session history with BrowserRouter.

Our application will dynamically generate the pages that React.js renders for us in the browser. BrowserRouter will access the session history within our browser, so that we can let our users navigate through our application’s project structure with the routes that we define using the <Router> component.

## Organize your App in a Container

To better organize our logic within the structure of our App, an ideal approach is to create a container to hold the top-level component that renders our wallet. This makes it possible to more easily use flexbox to design the components of our application so that they can be viewed on mobile devices.

This primary node that is the main container of our App, will also hold the primary components that will render the logic we need for our wallet. Before we can do that we should list a few of the views that we will need to implement to make this successful:

1. Create Invoice/Transaction
2. View Transactions
3. Pay Transaction
4. View History

The first thing we have to build is a navigation bar so that we can organize all of the views we need our users to work with. React-Bootstrap makes this really easy with import Navbar from "react-bootstrap/Navbar"; as shown below.

We have to make some changes to the default App.js file originally deployed by create-react-app to make this possible. Take a look at the source code below that we need to use in the new version of our App.js file.

/\*  
 Pay Me Now Wallet  
 P2P anonymous payments  
\*/  
  
import React, { Component } from 'react'; // Added Component  
// Use Link (see r-r-d docs here), for ref to home without refresh  
import { NavLink, withRouter } from "react-router-dom";  
// Import navbar component given to you by bootstrap   
import Navbar from "react-bootstrap/Navbar";  
import Nav from "react-bootstrap/Nav";  
import NavDropdown from "react-bootstrap/NavDropdown";  
import './App.css';  
  
// We need a component to "contain" our App  
class App extends Component {  
 // Need to render App container  
 render() {  
 return (  
 // should probably discuss className syntax in article  
 <div className="App container">  
 { /\* Add bootstrap Navbar \*/ }  
 { /\* probably best to discuss JSX and JSX comments \*/ }  
 { /\* collapseOnSelect - Toggles expanded to false after the onSelect  
 event of a descendant of a child <Nav> fires. \*/ }  
 <Navbar bg="light" expand="lg">  
 <Navbar.Brand as={ NavLink } to="/">  
 MyPay Wallet  
 </Navbar.Brand>  
 <Navbar.Toggle aria-controls="basic-navbar-nav" />  
 <Navbar.Collapse id="basic-navbar-nav">  
 <Nav className="navigate">  
 <Nav.Link href="#register">  
 Register  
 </Nav.Link>  
 <Nav.Link href="#login">  
 Log In  
 </Nav.Link>  
 </Nav>  
 </Navbar.Collapse>  
 </Navbar>  
 </div>  
 );  
 }  
}  
export default App;

We are telling React that we want to use the Bootstrap4 fluid attribute and layout that is configured by default with <Navbar>. We also declare the primary div.container to implement a *responsive application container* with a fixed-width, so that we can add our navigation bar easily while ensuring its responsiveness. Next, we have to figure out a way to get our users back to the homepage without triggering a refresh event. To get our users to their homepage dashboard we are going to use React-Router’s as={ NavLink } component to get us there. We have to specify a path which in the case is the root of our project directory or, to="/".

We will use the Bootstrap4 <Navbar.Collapse> component to give us a responsive and collapsable *hamburger* icon on mobile devices for the two <Nav.Link> components will use for our future registration and login functionality.

Finally, we need to begin to style our application. Please copy the following css styling to the same directory as your App.js script in your project directory, and call the file App.css.

.App {  
 margin: 0px;  
 padding: 0px;  
}  
  
.App .navbar-brand {  
 font-weight: bold;  
}  
  
.container {  
 max-width: 100%;  
}

### Implement your landing page

With the primary application container ready, we need to implement our landing page in a Home container that we can use to respond to the / route where our primary components will live, and that we can use to make requests to our serverless + microservices, using *asynchronous* calls to our *services in the cloud*. We will want to differentiate our primary containers from the rest of the components in our application, and we will use $ mkdr src/containers from our project’s root directory to save these files.

Now that we have structured our application properly, we can complete the first iteration of our landing page so that we can show something to the rest of the world. Create a new file called Home.js inside of the src/containers project directory.

All of the primary containers that provide our routes with a response, and that will send requests to our serverless backend, are just the primary views that our user interacts with in our application. Take a look at the file we have implemented for you in the src/containers/Home.js project file, and make the changes in your project as needed. Below is what we completed for this tutorial:

import React, { Component } from "react";  
import "./Home.css";  
  
export default class Home extends Component {  
 render() {  
 return (  
 <div className="Home">  
 <div className="landing-page">  
 <h1>UPay Simple. A Secure Wallet</h1>  
 <p>  
 A simple & Anonymous P2P payments app.  
 </p>  
 </div>  
 </div>  
 );  
 }  
}

Slowly we make progress. Here you have rendered your first html page using JSX, an extension to JavaScript that lets you use an xml-like syntax in the views that you implement to elegantly describe what you want your user’s interface to look like.

Here is the magic that we find in React; it embraces the fact that the logic required to implement an elegant user interface is tightly coupled with rendering, state changes, data preprocessing and application logic also. React allows you to create simple and reusable components that let you separate architectural considerations efficiently, to help you feel more comfortable writing your html markup into your JavaScript functions, so that you can more easily render the views you need to your users.

The JSX we are rendering above is only going to display the first version of our homepage to the users of our new Wallet. Copy the css styling below into the src/containers/Home.css file in your project directory so we can start making this page look presentable:

.Home .landing-page {  
 text-align: center;  
 padding: 65px;  
}  
  
.Home .landing-page h1 {  
 font-weight: 700;  
 font-family: sans-serif, "Roboto";  
}  
  
.Home .landing-page p {  
 color: #999999;  
}

### Configure your Application’s Routing

Now that we have a Home page component that we can use to start displaying some form of content to our user, we need to configure the routes that our application will use so that the view implemented above, can respond appropriately when a user triggers an event that calls the / route from anywhere in the application. We will use react-router and the Switch component it gives us, to render an *exclusive* route that we can display to the user as the first resource defined within the component, that precisely matches the path we need.

Review the code we implemented below that you will need to create as a new src/Routes.js file in the project directory, and add the following to your implementation:

import React from "react";  
import { Route, Switch } from "react-router-dom";  
import Home from "./containers/Home";  
  
export default () => <Switch>  
 <Route path="/" exact component={ Home } />  
 </Switch>;

Taking a look at this a bit deeper, you can see that we are using the exact property because we **must** precisely match the / path defined as a route in our implementation file. Using the Switch component requires us to use the exact prop like this because it will force the / route to match every route that uses the / at the start of its declaration.

### Display your Home Container and Declare both Registration/Login Routes

We need to refactor our App.js component so that it knows how to properly respond to the events triggered by a user who is requesting a particular route from our application. We need to add the <Routes /> component that we implemented above so that our user can navigate through the different features that we will provide them with to make and accept P2P payments from within our application. Take a look at the updated App.js source code below that you need to be sure to implement in your own project:

import React, { Component } from 'react'; // Added Component  
// Use Link (see r-r-d docs here), for ref to home without refresh  
import { Link } from "react-router-dom";  
// Import navbar component given to you by bootstrap   
import { Navbar } from "react-bootstrap";  
// Import Routes component so that App knows how to   
// respond with correct view  
import Routes from "./Routes";  
import './App.css';  
  
// We need a component to "contain" our App  
class App extends Component {  
 // Need to render App container  
 render() {  
 return (  
 // should probably discuss className syntax in article  
 <div className="App container">  
 { /\* Add bootstrap Navbar \*/ }  
 { /\* probably best to discuss JSX and JSX comments \*/ }  
 { /\* collapseOnSelect - Toggles expanded to false after the onSelect  
 event of a descendant of a child <Nav> fires. \*/ }  
 <Navbar fluid collapseOnSelect>  
 <Navbar.Header>  
 <Navbar.Brand>  
 <Link to="/">  
 Pay My Wallet  
 </Link>  
 </Navbar.Brand>  
 </Navbar.Header>  
 </Navbar>  
 <Routes />  
 </div>  
 );  
 }  
}  
export default App;

You should now have a homepage that looks something like the image below:

**myPay first Homepage**

![alt text](data:text/html; charset=utf-8;base64,)

alt text

We have setup our application’s first *route* that we can use to always get our user back to the homepage. It would probably be a good idea to take a look at the two links that we initially setup to use for our future Login and Registration routes that we are going to use to get our users authenticated so that we can assign them the permissions they need to use the different resources we will build into the functionality of this project.

Take a second to refactor your src/App.js file so that your render method looks like this:

render() {  
 return (  
 // should probably discuss className syntax in article  
 <div className="App container">  
 <Navbar bg="light" expand="lg">  
 <Navbar.Brand as={ NavLink } to="/">  
 MyPay Wallet  
 </Navbar.Brand>  
 <Navbar.Toggle aria-controls="basic-navbar-nav" />  
 <Navbar.Collapse id="basic-navbar-nav">  
 <Nav className="navigate">  
 <Nav.Link as={ NavLink }  
 to="/register"  
 className="navi-link"  
 exact>  
 Register  
 </Nav.Link>  
 <Nav.Link as={ NavLink }  
 to="/login"  
 className="navi-link"  
 exact>  
 Login  
 </Nav.Link>  
 </Nav>  
 </Navbar.Collapse>  
 </Navbar>  
 <Routes />  
 </div>  
 );  
}

The *routes* declared in the <Nav.Link> components rely on the import { NavLink } from "react-router-dom"; statement we are also using for the home page route. If you click on the Register or Login button on your homepage you will see that the URL in the Address Bar reflect the slection chosen by the user, either http://localhost:3000/register or http://localhost:3000/login, respectively.

Right now there is nothing that react can render to the page when the links are selected by the user since there is no registration, login, or any page for that matter. In case a user decides to request a *route* that we dont have an implementation for, we should make it so that our application responds with a proper *404 Page Does Not Exist* response.

### Handle 404 Responses and Routing

With react-router we can see that it is pretty easy to declare and implement new routes for your application as needed. We still need to create the components that the *routes* will point to. As our application’s **permissions-model** grows in complexity, the *Higher Order Components (HOC)* that we will use to wrap our *routes* with, will also grow in complexity as we use them to properly authenticate the routes that we will implement with you in this article.

We need to start off by implementing a component that we can save as src/containers/404Page.js so that we can create the view we’ll need React.js to display for us in-app if and when a user tries to access a resource that does not exist.

Add the following code to the new 404Page.js file that we created and I will show you how to add it as a default *catch-all* route in case our users need it:

import React from "react";  
import "./404Page.css";  
  
export default () => <div className="Page404">  
 <h3>Sorry, this page does not exist within the MyPay Platform... yet!</h3>  
 </div>;

The only thing that this component is responsible for is to display a message to our user informing them that the resource that is being accessed does not exist.

We can style this page a bit further and just for a bit more practice with the .css code I have left for you below:

.Page404 {  
 padding-top: 100px;  
 text-align: center;  
}

The styling above is just aligning the output message our component is displaying to the user in the center of the page using a 100px padding from the top of the page

#### Catching All Unmatched Routes

We can now use react-router to implement the route that we need to handle the 404 Page request our users will inevitably generate for our serverless…servers. Did I say that right? Either way, now we need to tell our app when and how to show our users our new 404Page.js component.

We need to implement a new <Route /> component in the src/Routes.js file located in the root project directory. We need to add a new case to the <Switch> block that needs to look a little something like this:

import React from "react";  
import { Route, Switch } from "react-router-dom";  
import Home from "./containers/Home";  
import Page404 from "./containers/404Page";  
  
export default () => <Switch>  
 { /\* This is our home page route for the main landing page to the app \*/ }  
 <Route path="/" exact component={ Home } />  
  
 { /\* This route will catch all unmatched routes && MUST BE LAST!!! \*/ }  
 <Route component={ Page404 } />  
 </Switch>

The new case we are adding to handle our Page 404 error responses must always be included in the <Switch> block last. Our Page404 component will only handle requests that have failed to be handled by any of the initial routes that we have implemented.

Once you have this complete you can now click on the login or register buttons in the app that should now take you to a 404 page response similar to what you can see below:

![alt text](data:text/html; charset=utf-8;base64,)

alt text

Now that we have a bit of a *front-end infrastructure* in place, we can now work on figuring out how to configure the AWS Services that we have implemented to support this app and its users. Let’s tae a look how we can start leveraging cloud based tools like AWS-Amplify to easily give our app what it needs to do something amazing like *making the world a better place*…

## Building a Modern Web Application with AWS-Amplify

Traditional web-architectures using a *client-server* approach lead to a poor user experience because of the need to re-render dynamic pages for every click of a button that needs to wait for an *asynchronous* response from a server. On the contrary, Modern Web Applications use a *Single Page Application* approach to package all of the application components into a single *static* file to offer a *native-like* feel to the user using brebuilt layout and JavaScript files to execute backend logic without having to re-render the page content.

AWS-Amplify just provides everything that we need to implement and deploy scalable cloud based apps that uses a *CLI* and *library* that really simplifies the development of web and mobile applications. In the next article we will even go over and show you how to deploy your app onto an infinite amount of stages by using AWS-Amplify.

### Install and Configure Amplify

We need to be able to communicate with AWS Services like Cognito, S3, API Gateway, and others like DynamoDB. The first thing we have to get done is to install the Amplify library thats going to let us use all of these cloud based tools really easily:

$ npm install aws-amplify --save

Once this completes double check that Amplify is now a listed dependency in your package.json file located in the root of your project directory.

When we deployed our *serverless + microservices*, we were shown a few different *Stack Outputs* provided by AWS that gave us the unique resource names and ID’s that we now need to use to configure AWS Amplify. With the information you gather, we need to then proceed to create a new file called src/config.js so that we can add the details shown below:

export default {  
 s3: {  
 REGION: "S3\_BUCKET\_REGION",  
 BUCKET: "S3\_BUCKET\_NAME"  
 },  
  
 apiGateway: {  
 REGION: "API\_GATEWAY\_REGION",  
 URL: "API\_GATEWAY\_URL"  
 },  
  
 cognito: {  
 REGION: "COGNITO\_REGION",  
 USER\_POOL\_ID: "COGNITO\_USER\_POOL\_ID",  
 APP\_CLIENT\_ID: "COGNITO\_APP\_CLIENT\_ID",  
 IDENTITY\_POOL\_ID: "COGNITO\_IDENTITY\_POOL\_ID"  
 }  
};

Each of the values shown in the file above needs to be replaced with the stack outputs your were displayed when you deployed your backend services on AWS in the previous chapters of this tutorial.

You can find these stack outputs by looking through the details of the build logs in your CodePipeline implementation that we completed for your CI/CD.

### Implement AWS Amplify

To complete the configuration and implementation of Amplify with our application we need to first add the folllowing statements to the top of src/index.js:

import Amplify from "aws-amplify";

also include: import config from "./config";

Finally, you need to provide Amplify with a mapping of the config parameters we implemented above and this will allow us to initialize Amplify with everything we need to have our app use the services we need on the cloud. Our implementation of src/index.js needs to look like the following file below:

import React from 'react';  
import ReactDOM from 'react-dom';  
import './index.css';  
import App from "./App";  
import { BrowserRouter as Router } from "react-router-dom";  
import \* as serviceWorker from "./serviceWorker";  
import Amplify from "aws-amplify";  
import config from "./config";  
  
Amplify.configure({  
 Auth: {  
 mandatorySignIn: true,  
 region: config.cognito.REGION,  
 userPoolId: config.cognito.USER\_POOL\_ID,  
 identityPoolId: config.cognito.IDENTITY\_POOL\_ID,  
 userPoolWebClientId: config.cognito.APP\_CLIENT\_ID  
 },  
 Storage: {  
 region: config.s3.REGION,  
 bucket: config.s3.BUCKET,  
 identityPoolId: config.cognito.IDENTITY\_POOL\_ID  
 },  
 API: {  
 endpoints: [  
 {  
 // NOTE: The API "name" is critical and used by aws-amplify  
 // when an amplify API.post() method where the first  
 // argument is the name of this API field!!!  
 name: "invoice-log-api",  
 endpoint: config.apiGateway.URL,  
 region: config.apiGateway.REGION  
 }  
 ]  
 }  
});  
  
ReactDOM.render(  
 <Router>  
 <App />  
 </Router>,  
 document.getElementById("root")  
);  
  
// If you want your app to work offline and load faster, you can change  
// unregister() to register() below. Note this comes with some pitfalls.  
// Learn more about service workers: https://bit.ly/CRA-PWA  
serviceWorker.unregister();

You have to keep in mind that in the context above Amplify refers to to the Cognito Authentication service on the cloud as Auth. The same holds true form both the S3 Upload service and API Gateway; Storage refers to S3 while API refers to the later.

The configuration settings also let us add endpoints that our app can communicate with via API Gateway. The name of the enpoint we deployed with Lambda on the ServerlessFramework is the invoice-log-api. If you notice here we have a data structure that has an array of *endpoint objects*. Our app currently only has one endpoint, but as we continue to build it out here is where we will be configuring each of the endpoints we need to use to access the services we deploy on API Gateway.

Lastly, me make use of the mandatorySignIn flag by setting it to true inside of the Auth object so that we can force our users to register and sign in with Cognito before they use any of our application’s resources.

Now that we have all of our application’s plumbing in place so that our left hand knows where our right hand is we can implement a simple user registration and login feature that will let us authenticate everyone who wants to use our app!

## Implement the Sign-In Interface

Eventually our user will have some form of credential that they can use to sign-in to our application, and we have chosen to begin with the implemention of the view for this interface to help demonstrate how we can use React.js to make *asynchronous calls* to the Cognito service on the AWS Cloud. When we develop our registration interface using forms in bootstrap4, we will dive deeper into the use of making *async* calls to let our user sign-in to our app with the email attributes that we configured as a username with Cognito, using Infrastructure As Code on our serverless backend.

### Create A Login Component Hierarchy and Route

We need to setup a simple methodology to manage our state properties in react to store the requested credentials from our user. Add a new file called src/containers/Signin.js to your project directory and implement the code below:

import React, { Component } from "react";  
import Button from 'react-bootstrap/Button';  
import Form from 'react-bootstrap/Form';  
import { Auth } from "aws-amplify";  
import "./Signin.css";  
  
export default class Signin extends Component {  
 constructor(props) {  
 super(props);  
  
 this.state = {  
 email: "",  
 password: ""  
 };  
 }  
  
 validateForm() {  
 return this.state.email.length > 0 && this.state.password.length > 0;  
 }  
  
 handleChange = event => {  
 this.setState({  
 [event.target.id]: event.target.value  
 });  
 }  
  
 // Use async promise to wait for response from aws-amplify api  
 handleSubmit = async event => {  
 event.preventDefault();  
  
 // Amplify Authentication logic  
 try {  
 // Make call to Auth API using aws-amplify  
 await Auth.signIn(this.state.email, this.state.password);  
 alert("Logged In!");  
 } catch ( err ) {  
 alert(err.message);  
 }  
 }  
  
 render() {  
 return (  
 <div className="Signin">  
 <Form onSubmit={ this.handleSubmit }>  
 <p>  
 <strong>Returning Users Please Sign-in.</strong>  
 </p>  
 <Form.Group controlId="email">  
 <Form.Label>  
 Email address  
 </Form.Label>  
 <Form.Control autoFocus  
 size="lg"  
 type="email"  
 placeholder="Enter email"  
 value={ this.state.email }  
 onChange={ this.handleChange } />  
 <Form.Text className="text-muted">  
 We will never share your private information with a third-party.  
 </Form.Text>  
 </Form.Group>  
 <Form.Group controlId="password">  
 <Form.Label>  
 Password  
 </Form.Label>  
 <Form.Control size="lg"  
 type="password"  
 placeholder="Password"  
 value={ this.state.password }  
 onChange={ this.handleChange } />  
 </Form.Group>  
 { /\* FIXME: Get Remember me check working so that it grabs user email from cookie \*/ }  
 <Form.Group controlId="rememberUser">  
 <Form.Check type="checkbox" label="Remember Me" />  
 </Form.Group>  
 <Button block  
 size="lg"  
 disabled={ !this.validateForm() }  
 variant="primary"  
 type="submit">  
 Login  
 </Button>  
 </Form>  
 </div>  
 );  
 }  
}

The first thing we have to pay attention to here is the implementation of the state object inside of our component’s constructor. We use state attributes to store the user’s email and password data that we can access using this.state.email and this.state.password as the value in our sign-in form fields that will be updated by React.js on every change of state. React.js literally provides you with a canvas to create your front end application on. As you change the state of your application’s attributes, React.js will display a new render, or the most updated version of your application.

It is important to understand this nature of React.js, and how it perpetually renders, and re-renders your page to display the most recent state changes that your application has undergone. If you try to change the state of an attribute within a render() function you will inevitably get caught in one hell of an infinite-loop. You need to change the state of your attributes with an event handler like the one we implemented above called handleChange().

handleChange() is triggered by an application event and uses the controlId declared in each <Form.Group> in the input fields shown. It proceeds to update the state of each attribute when the user enters the appropriate data.

Form validation is always a good practice and looking a bit deeper we can see that the function called validateForm checks to ensure that our email and password fields are not empty prior to the execution of the Auth.signIn() method.

When the state of the application is ready to submit data to our services on the cloud, we then use Amplify to call our resources in Cognito to authenticate a user. Take a look at the *asynchronous* promise returned by the handleSubmit() function shown above. We are using our state sttributes to call Amplify’s Auth.signIn() method to validate our credentials and authenticate our user. Using the await feature to return a promise, we execute Auth.signIn() to securely sign our user into the app.

Next, in our src/containers/Signin.css file we can style our sign-in screen accordingly:

@media all and (min-width: 480px) {  
 .sign-in {  
 padding: 60px 0;  
 }  
  
 .sign-in form {  
 margin: 0px auto;  
 max-width: 320px;   
 }  
}

We need to complete the implementation of our sign-in page by adding a *Sign-In* route to our src/Routes.js file below our root path. Your new file should look like the source below:

/\*  
 Pay Me Now Wallet  
 P2P anonymous payments  
 Routes.js  
\*/  
  
import React from "react";  
import { Route, Switch } from "react-router-dom";  
import Home from "./containers/Home";  
import Signin from "./containers/Signin";  
import Page404 from "./containers/404Page";  
  
export default () => <Switch>  
 { /\* This is our home page route for the main landing page tyo the app \*/ }  
 <Route path="/" exact component={ Home } />  
 { /\* This is the Login Route \*/ }  
 <Route path="/login" exact component={ Signin } />  
 { /\* This route will catch all unmatched routes && MUST BE LAST!!! \*/ }  
 <Route component={ Page404 } />  
 </Switch>

With that all setup and ready to go, if you look at your browser now at http://localhost:3000/login you should see a page that looks a little something like the image below:

![alt text](data:text/html; charset=utf-8;base64,)

alt text

And thats it. You have implemented everything we need to set up the skeleton for a simple React.js application. In the next article we will discuss how you will need to actually connect this new front end to all of the serverless + microservices you built on AWS Lambda. We will finish off the next article with showing you how to deploy all of these new features that take advantage of your GeneralLedger in the cloud on AWS Amplify.

### You completed the implementation of your React.js Framework. Good Luck!

## Part 7: Integrate your serverless + microservices with React.js and deploy your app on AWS-Amplify.

* [Part 7: Integrate React with Serverless and Deploy on Amplify](https://github.com/lopezdp/TechnicalArticles/blob/master/ServerlessReactDeployedOnAmplify.md) - *Not Published.*

…