# Part 7: Integrate your serverless + microservices with React.js and Implement Multi-Step Registration

*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)
* [Part 6: Build a React.js Frontend for your PayPal clone](https://github.com/lopezdp/TechnicalArticles/blob/master/BuildAReactJsFrontendforYourPayPalClone.md) - *Not Published.*

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 continue to cover the needed 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

## Implement User Permissions and Registration

Now that this application of ours has a login interface that our users can use to authenticate themselves, we are going to have to make sure that there is a mechanism in place to enable the permissions our users need to create and manage their transactions within our app. To properly complete these configuration steps we will need to be able to load our application’s current state from our user’s login session managed by Cognito, we’ll need to implement a few redirects for proper login and logout functionality, and more importantly we always have to make sure that we are giving our users good feedback when logging in so that they can be sure that they are submitting the correct credentials.

Initially we really need a way to update the application state when the user signs into our app. This brings up the concept of *“Lifting”* state up to a *Parent* component so that the login state can be passed down as a *prop* to other components that will share this attribute later. Application state can only be passed down from Parent to Child component. As the complexity of the application grows, state management should be a top priority so that you can architect your UI accordingly.

Our src/App.js file is what we will use as the Parent component for our application to manage a simple authentication scheme; and we will pass down the authentication state of the app as a prop to the remaining Child components that we will implement to deliver the functionality of our wallet. We will create a handler function that will allow us to determine if a user is authenticated by setting the user’s authentication state in our Parent component or the src/App.js file that we are using to render our application’s UI components.

Using a flag to determine the user’s authentication state will allow us to implement a function to change the state of the flag every time a user signs into, or out of our application. To initialize our flag that we are using as a state attribute with the function that we will create to update the state changes we need to account for when our users login or out, we will need to pass a reference to this function to the <Signin /> component that will use it to update the state attribute in src/App.js on user login or logout.

### Implement Routes with Access to the App Session

With Amazon Cognito we can access the session state declared in our Parent component, and we can pass it into the Child component created by the application for each of our routes. Using the appropriate property attributes or props in our case, copy and paste the following object that you need to add into your src/App.js file just below your render() function to store the authentication state for each user that signs into our app.

const childProps = {  
 isAuthenticated: this.state.isAuthenticated,  
 userHasAuthenticated: this.userHasAuthenticated  
 };

Having declared an object that will hold the value of the authentication state that you need to pass into each rendered component used by your application, you now need to pass the object and its values into the <Route /> component that is declared at the end of the render() function in your src/App.js file. The declaration of the route component should now look like this: <Routes childProps={ childProps } /> and the entire updated src/App.js file needs to look like this:

/\*  
 myPay 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 navbar component given to you by bootstrap   
import { NavLink } from "react-router-dom";  
import Navbar from "react-bootstrap/Navbar";  
import Nav from "react-bootstrap/Nav";  
import NavDropdown from "react-bootstrap/NavDropdown";  
import Routes from "./Routes";  
import './App.css';  
  
// We need a component to "contain" our App  
class App extends Component {  
 constructor(props) {  
 super(props);  
 this.state = {  
 isAuthenticated: false  
 };  
 }  
  
 userHasAuthenticated = authenticated => {  
 this.setState({  
 isAuthenticated: authenticated  
 });  
 }  
  
 // Need to render App container  
 render() {  
  
 const childProps = {  
 isAuthenticated: this.state.isAuthenticated,  
 userHasAuthenticated: this.userHasAuthenticated  
 };  
  
 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 childProps={ childProps } />  
 </div>  
 );  
 }  
}  
export default App;

The fact is we still have plenty of work to get our application to actually do something we these values now. Our routing mechanism with react-router has to initialize the Child component that the user requests with these props, and in our first use-case we will use them with our Signin component so that we can set our application’s initial authentication state, just to get this thing finally working!

#### Project Structure and HOC’s

Let’s take a moment to talk about an advanced technique used in React.js as an emergent design pattern, because of its ability to allow developers to implement composite functions. Yes a function as an input to another function, or in the case of *Higher Order Components* in React.js, **a function that accepts a Component as an argument to return a new Component** that we canleverage to reuse Component logic. We need to use this concept to implement an *HOC* that will create a new Route for us that renders a new Child Component that will have the property values that have to be stored for use in the UI to deal with the appropriate state changes triggered by our users.

Ideally we should define an algorithm to follow for the implementation of our use case and its *HOC* that we will use to give us the Child component we need with all of the properties we are going to use to trigger an optimal user experience. The tool we are designing should be able to execute the following series of actions:

1. The *HOC* will output a new <Route /> component that will include all of the properties neded by the child component rendered by the app. We will use a property, component to render the appropriate route that is found by its reference to the component that we want shown in the UI. The childProps object we declared in src/App.js will be passed into the component that we output from our *HOC*.
2. We will also create a render method that we can pass into the Route component that will output from our *HOC* instead of a component, to better control what we pass into our route components. Our *HOC* will return a Route by accepting both a component and childProps as props inputs to enable us to provide the component and the property values that have to be rendered to the UI.
3. We will declare an object that will will pass into our function that will define the values of our component as C and the properties that we want to set and render inside of our Route as cProps. We will pass a render function also that will output the new component we desire.

Below is the implementation of the *HOC* that we defined above:

import React from "react";  
import { Route } from "react-router-dom";  
  
export default ({  
 component: C,  
 props: cProps,  
 ...rest  
}) => <Route  
 { ...rest }  
 render={  
 props => <C { ...props } { ...cProps } />  
 }  
 />;

The file for the code above should be organized in a new directory that we will call src/components to help us maintain a separation of concerns between React.js component that we will use to make asynchronous calls to our serverless + microservices and the code that we write to render a beautiful UI. Save the file above as src/components/AppliedRoute.js and let’s move on so I can show you how this thing works.

Since we will use this implementation to dynamically provide us with the routes and properties that we need for each component that we will render to our user, we will also need to refactor our existing src/Routes.js file so that we can use the new childProp values that we are passing into each of the new components that our new Route component will output to the UI.

Your new src/Routes.js file has to look like the source code you see here:

/\*  
 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";  
import AppliedRoute from "./components/AppliedRoute";  
  
export default ({childProps}) => <Switch>  
 { /\* This is our home page route for the main landing page tyo the app \*/ }  
 <AppliedRoute path="/"  
 exact  
 component={ Home }  
 props={ childProps } />  
 { /\* This is the Login Route \*/ }  
 <AppliedRoute path="/login"  
 exact  
 component={ Signin }  
 props={ childProps } />  
 { /\* This route will catch all unmatched routes && MUST BE LAST!!! \*/ }  
 <Route component={ Page404 } />  
 </Switch>

Now we are using an AppliedRoute component everywhere that you need to set a series of childProps that we will use in our Child component, in our case we are accessing the userHasAuthenticated prop that manages the authentication state of our application from the src/App.js Parent component. To change the appliation state in our Parent component src/App.js, we have to trigger the userHasAuthenticated(bool) function that we declared in src/App.js that we passed into our AppliedRoute as a childProp.

In our Signin component our users will interact with the functionality that will log them into and out of our application. Inside of /src/containers/Signin.js, we need to make sure we have implemented the following line of code: this.props.userHasAuthenticated(true).

This line of code has to executed after the call to signin with Cognito is made with: await Auth.signIn(this.state.email, this.state.password); inside of your handleSubmit() funtion that we will trigger when the login form is submitted by the user.

Your new handleSubmit() async function inside of your src/containers/Signin.js component should now look like this:

// 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);  
 this.props.userHasAuthenticated(true)  
 } catch ( err ) {  
 alert(err.message);  
 }  
}

To execute this functionality we now have to implement a UI element that will let the user fire off these events. We need a sign out button and we need one quick.

#### Using a Button Element to Sign Off

We are now ready to start adding a few more interactive UI elements into the layout of the app. We need to apply the authenticate state functionality so that we can let our users interact with our app on Sign Out. When a user is authenticated, they will need to see a button that will let them sign out of our application also. We will accomplish this in React.js using a <Fragment> component. A Fragment is great for creating groups of child elements without cluttering the DOM full of extra nodes.

In our case we need an option to display two links in the UI when a user is not authenticated; we should display a *Register* and a *Sign In* link. In the case of an authenticated user, we need to only display a *Sign Off* link in the UI. We will accomplish this using a ternary operator and a Fragment component. When this.state.isAuthenticated is true, we will display the Sign Off link to the user. When the isAuthenticated flag is flase out ternary operator will return our new Fragment component where we will use JSX to declare our Register and Sign In links to the user.

Make sure that the source code in your src/App.js file looks like the following file, while making sure to add Fragment to your list of react imports as shown below:

/\*  
 myPay Wallet  
 P2P anonymous payments  
\*/  
  
import React, { Component, Fragment } from 'react'; // Added Component  
// Use Link (see r-r-d docs here), for ref to home without refresh  
// Import navbar component given to you by bootstrap   
import { NavLink } from "react-router-dom";  
import Navbar from "react-bootstrap/Navbar";  
import Nav from "react-bootstrap/Nav";  
import NavDropdown from "react-bootstrap/NavDropdown";  
import Routes from "./Routes";  
import './App.css';  
  
// We need a component to "contain" our App  
class App extends Component {  
 constructor(props) {  
 super(props);  
 this.state = {  
 isAuthenticated: false  
 };  
 }  
  
 userHasAuthenticated = authenticated => {  
 this.setState({  
 isAuthenticated: authenticated  
 });  
 }  
  
 // Update authentication state on signout event  
 handleSignOut = async event => {  
 this.userHasAuthenticated(false);  
 }  
  
 // Need to render App container  
 render() {  
  
 const childProps = {  
 isAuthenticated: this.state.isAuthenticated,  
 userHasAuthenticated: this.userHasAuthenticated  
 };  
  
 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">  
 { this.state.isAuthenticated  
 ? <NavDropdown.Item onClick={ this.handleSignOut } className="navi-link">  
 Sign Out  
 </NavDropdown.Item>  
 : <Fragment>  
 { /\* Fragment is like placeholder component \*/ }  
 <Nav.Link as={ NavLink }  
 to="/register"  
 className="navi-link"  
 exact>  
 Register  
 </Nav.Link>  
 <Nav.Link as={ NavLink }  
 to="/signin"  
 className="navi-link"  
 exact>  
 Login  
 </Nav.Link>  
 </Fragment> }  
 </Navbar.Collapse>  
 </Navbar>  
 <Routes childProps={ childProps } />  
 </div>  
 );  
 }  
}  
export default App;

Take a look at the new handleSignOut async function that we defined above too. When called by the UI after the user triggers the Sign Out event, our application will call handleSignOut and change the userHsAuthenticated flag to false so that our application’s authentication state can be updted to reflect the current state of the user’s session as needed. See the code below for a better look at the function in question:

// Update authentication state on signout event  
handleSignOut = async event => {  
 this.userHasAuthenticated(false);  
}

To check the implementation we have just worked through, we will need to create a user for ourselves from the command line or terminal since we dont have a mechanism to register a user with at the moment. Luckily for us, Amazon Cognito gives us a really easy way to do this using the following commands to create a user in Cognito that we can use to test our connections with.

##### Creating a Test User

Using an email address and a password we will run a few commands using the AWS-CLI that we installed in a previous chapter in this series. This user will also have to be confirmed so this will take few steps to complete. Below is the first command that you need to run from that terminal on your pretty Mac Book Pro.

$ aws cognito-idp sign-up \  
 --region us-east-1 \  
 --client-id 52sdl6jc2kk571gk2do3r39lol \  
 --username davidplopez@live.com \  
 --password Passw0rd!

Here is the Output you should see in your terminal after running the command above:

{  
 "UserConfirmed": false,  
 "CodeDeliveryDetails": {  
 "Destination": "d\*\*\*@l\*\*\*.com",  
 "DeliveryMedium": "EMAIL",  
 "AttributeName": "email"  
 },  
 "UserSub": "800ef60e-2e91-4354-9956-1b1199b3cdee"  
}

Just like you did in your own config.js file, here you will need to do the same and use your own client-id and region values based on the *Service Information* and *Stack Outputs* you received when you deployed your service on CodePipeline in the previous chapter on CI/CD. Furthermore, before we can use this username to authenticate ourselves within our application, we will need to verify this user in Cognito with the fopllowing command:

$ aws cognito-idp admin-confirm-sign-up \  
 --region us-east-1 \  
 --user-pool-id us-east-1\_IJ3kOWibj \  
 --username davidplopez@live.com

With this step complete we can now test our Fragment component to see if our Sign Out link is displayed to our users when they authenticate themselves against Cognito.

The following image is what you should be able to see on your own local environment (Notice the new Sign Out link that appears after the new test user signs into the app!):

A screenshot of a social media post

Description automatically generated

### Loading the Application State from its Session

Typically we can make use of Cookies or LocalStorage to store the user’s sign-in data that we can load from the session stored in the browser. In the case of a *Progressive Web Application* we can use these tools to persist *offline data* to allow our users to work with our app in a native-like environment. With AWS-Amplify we can store our session information automatically and use Amplify to load the session information we need when a user signs in, into the application’s state.

We are going to work with the Auth.currentSession() method provided by Amplify to return a promise that we can resolve into a session object that we use to verify a user’s login state. We will implement a componentDidMount function in our React.js app to load our current session *asynchronously*.

We will use *promises* implemented with the aync, await pattern to load our application only after our promise returns an authenticated user object after the complete execution of the logic in our componentDidMount function. We need a few state attributes that we can use as flags to tell our application when a user isAuthenticated and that our application isAuthenticating a user on login.

Implementing Auth.currentSession() will load a user’s session when they log in, and our application will update our isAuthenticating flag if the session loads successfully. Amplify will throw an error that we will handle with a try/catch method if there is no user signed into the application. Our componentDidMount inside of our src/App.js file should look like the following source code:

async componentDidMount() {  
 try {  
 await Auth.currentSession();  
 this.userHasAuthenticated(true);  
 }  
 catch(e) {  
 if(e !== 'No current user'){  
 alert(e);  
 }  
 }  
  
 this.setState({ isAuthenticating: false });  
}

Dont forget to include: import { Auth } from "aws-amplify"; so that Amplify knows you’re trying to grab the session object from its grasp!!!

Leveraging the asynchronous pattern that we implemented above we need to wait for the application state to load with the user’s session we are taking from Amplify to render the app in an authenticated or unauthernitcated state, depending on the response we get back from Cognito and Amplify.

Below is what the new render function should look like that will let us conditionally render our views based on the authenticated status of the user.

// Need to render App container  
render() {  
  
 const childProps = {  
 isAuthenticated: this.state.isAuthenticated,  
 userHasAuthenticated: this.userHasAuthenticated  
 };  
  
 return (  
 !this.state.isAuthenticating &&  
 // 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">  
 { this.state.isAuthenticated  
 ? <NavDropdown.Item onClick={ this.handleSignOut } className="navi-link">  
 Sign Out  
 </NavDropdown.Item>  
 : <Fragment>  
 { /\* Fragment is like placeholder component \*/ }  
 <Nav.Link as={ NavLink }  
 to="/register"  
 className="navi-link"  
 exact>  
 Register  
 </Nav.Link>  
 <Nav.Link as={ NavLink }  
 to="/signin"  
 className="navi-link"  
 exact>  
 Login  
 </Nav.Link>  
 </Fragment> }  
 </Navbar.Collapse>  
 </Navbar>  
 <Routes childProps={ childProps } />  
 </div>  
 );  
}

A user can now log into our application and we will have to implement the functionality they will use to log out next.

### Create the Session when a user Signs On/Off & Redirect

// Update authentication state on signout event  
handleSignOut = async event => {  
 await Auth.signOut();  
 this.userHasAuthenticated(false);  
}

Amplify acts as our LocalStorage mechanism to help us persist our user’s session. With the code above our app will now clear the application’s state and clear the session from LocalStorage with Auth.signOut(). We need to add this feature to our handleSignOut function in our srcs/App.js file as shown above. With the implementation a user can logout and refresh the page, and Amplify will handle clearing our users session and application state to be sure to be completely signed out of our application.

The best way to handle a user signing out of our application is to send, or *redirect* our happy use-case actor back to the sign in page after the session is cleared and the app logs them out of the system. Conversely, when a user authenticates against our implementation of AWS Cognito, our app will have to make sure the user ends up on the home page after signing into our platform. Using the history object we can manipulate our browser’s *session history* to help our user navigate to the correct path in our app.

Thankfully, react-router gives us a method called history.push that we can use to find the right attribute within the history object that will let us push our user to the correct authenticated route on login. With our Login.js component that is rendered to our app with our <Route /> component, the new history object will be passed down into the Login component as a prop and we can easily redirect our user with the syntax you should already feel comfortable using in React.js:

* this.props.history.push("/");

To be able to access the *session history* that can direct our user correctly, we will use their location and push them to the root path of our project as determined by their authentication status. The component’s history property will be implemented like the source code below so that it is triggered correctly when the handleSubmit function is called:

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);  
 this.props.userHasAuthenticated(true);  
 // Implement hostory object here!  
 this.props.history.push("/");  
 } catch ( err ) {  
 alert(err.message);  
 }  
}

Take a look at your browser after your user signs into your application now, and you will see that after the user is authenticated by Amazon Cognito, your user will be redirected to the home page as intended. What happens when your user signs off? We will take a look at how to deal with that specific use-case next!

Unfortunately, the architecture of our application forces us to go a little outside of the box to implement our Sign Out functionality since the root application component, or our src/App.js file, is not actually rendered within a proper <Route />. We will need to use a [Higher Order Component (HOC)](https://facebook.github.io/react/docs/higher-order-components.html), and specifically, we will rely on an *HOC* called withRouter that is provided to us by react-router that will enable us to use the props that need to be accessible inside of src/App.js to pass our history object down into our Signin component. The properties will be available to you within the history object as determined by the HOC’s closest matching <Route>.

withRouter does not receive state change values to update the location within the hostory object, it only renders the page after every change in location that is persisted to the DOM from the <Router> component. withRouter will only re-draw wach page when its parent component is updated and is forced to be rendered as an update to the application’s state.

To implement withRouter correctly and to allow for the legitimate access of the history object’s session details, we first need to replace the last line in src/App.js with the following:

export default withRouter(App);

This is where we wrap our App component with the withRouter *Higher Order Component* so that we can share the history object and its session details with the rest of our App and its child components as properties that we can pass down into each with this wrapper. We just need to make sure that we are importing it correctly by using:

import { withRouter } from "react-router";

With everything in place to route and redirect our users correctly on sign-in and sign-out, we now need to ensure that our sign-out function looks like this:

handleSignOut = async event => {  
 await Auth.signOut();  
 this.userHasAuthenticated(false);  
 this.props.history.push("/signin");  
}

We should provide the user with a prompt of sorts to let them know what to expect from the program in development to account for the *asynchronous* requests made to our serverless + microservice, however, you should now see that the code above will now redirect the user of the application the the sign-in page upon signing off from our app.

### Prompt the user when Signing into the application

Using *asynchronous* methods to *get* a backend response from our serverless + microservices does introduce a bit of latency into the application that we will have to address so that our users will know that the app is working and has not crashed while waiting for the data requested. We will need to create a state attribute that we can use as a flag to determine when the application is *awaiting* an *asynchronous* response filled with data requested by the user from our UI. In our Signin component we will use an isLoading flag as a state attribute for the purposes we have just discussed. You need to implement your application’s initial state in the constructor of your Signin.js file as such:

this.state = {  
 isLoading: false,  
 email: "",  
 password: ""  
};

Finally, we need to make sure that we update our *asynchronous* handleSubmit function so that it updates the state of our application and the isLoading flag while the *asynchronous* request is processing so that we can use it to give a visual queue to our user so that they know that our application is working while they wait patiently… hopefully not for too long. Below is the completed handleSubmit function that you need to implement in your Signin.js component:

handleSubmit = async event => {  
 event.preventDefault();  
  
 this.setState({  
 isLoading: true  
 });  
  
 // Amplify Authentication logic  
 try {  
 // Make call to Auth API using aws-amplify  
 await Auth.signIn(this.state.email, this.state.password);  
 this.props.userHasAuthenticated(true);  
 this.props.history.push("/");  
 } catch ( err ) {  
 alert(err.message);  
  
 this.setState({  
 isLoading: false  
 });  
 }  
}

So this is great and all that we have a mechanism that changes the ephemeral state of our appliction using some esoteric state variable that we change dynamically on every user’s interaction with our application. But, how are we going to use it to let our user know that our application is in fact working, instead of just sitting there staring back blindly at our user?

#### Implement a UI Button with a dynamic Loading display

The easisest and most effective tool to use, is a button element that changes its behavior dynamically based on the state of the isLoading flag that we just implemented in the previous section. I decided to implement a component that we could just reuse so that we could better adhere to the infamous [DRY Principle](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself). I am certain we are going to need quite a few dynamic button elements that will tell the user when the app is loading new data from our serverless + microservices.

You need to start by creating a new file that I have called: src/components/UiLoadBtn.js and which needs to have the following code in its implementation:

import React from "react";  
import Button from "react-bootstrap/Button";  
import Spinner from "react-bootstrap/Spinner";  
import "./UiLoadBtn.css";  
  
export default ({ isLoading,   
 text,   
 loadingText,   
 className = "",   
 disabled = false,   
 ...props   
 }) => <Button className={ `LoaderButton ${className}` }   
 disabled={ disabled || isLoading }   
 {...props}>  
 { isLoading && <Spinner animation="grow" /> }  
 { !isLoading ? text : loadingText }  
 </Button>;

This is nothing more than a component that accepts a series of arguments that include the isLoading state attribute that we have declared for use as the flad that will let us dynamically change the way that this button appears to the user when the data is being processed by the application. The *JavaScript* object that we are defining as the parameters of our function also accept text that we will use to define what the button says to the user while it is in a *static* state, whereas the loadingText is what it will display to the user when in the dynamic state when isLoading is true.

Interestingly, I have decided to pass a new property that I have called disabled. disabled will let us decativate the user’s ability to interact with this *element* whenever we want, so that the user cannot click the button if the application is processing data *asynchronously* at any given moment. We do not want the user to trigger a second event while the first click is loading the original request’s data to the UI.

*Bootstrap4* and the *NPM* package built for its implementation in React.js with react-bootstrap is really convenient in that it provides us with a Spinner component that we can easily use to dynamically update the change of the button’s state with a visual representation of a *spinner* using this line of code:

* <Spinner animation="grow" />

There are quite a few options for the different visualizations you can use so make sure you visit the react-bootstrap docs to make sure you end up using the right element for your own app. Either way, *Bootstrap4* is great because with just the following bit of css our dynamic button implementation for our isLoading flag is now complete!

.UiLoadBtn {  
 margin-right: 7px;  
 top: 2px;  
}

While *Boostrap4* takes care of the display of our spinning *glyphicons*, we just need to implement our new component in our Signin.js component so that we can complete this task of letting our user know that the app is doing what it should be doing instead of, well… *buggin’ out*.

You will need to replace the <Button> element in your Signin.js file with the following:

{ /\* Loading Button Component \*/ }  
<UiLoadBtn block  
 size="lg"  
 disabled={ !this.validateForm() }  
 variant="primary"  
 type="submit"  
 isLoading={ this.state.isLoading }  
 text="Signin"  
 loadingText="Signing In..." />

Also, do not forget to correctly import this component into your Signin.js component so that you can use it. If you take a look at your browser you should be able to see the state of the button in the UI while the application processes your authentication request!

A screenshot of a cell phone

Description automatically generated

We need to add the ability to change a password and a few other features which we will go over in a separate section ater when we implement a Setting.js component. For now you should be able to see the Spinner component that Bootstrap4 gave us to load the next page on Signin. But what do we do to get a user credentialed and registered with our app anyway? We need a Registration component next!

### Implement a User Registration Workflow

The registration workflow we will create is quite simple, and will help us collect some basic information about our users so that we can take advantage of the AWS Amplify API and interact with Amazon Cognito to both confirm, and authenticate each user that registers with our app. We will confirm each user registration with Cognito’s API, which will send a *confirmation code* to every user’s email, and which they will need to enter into our two part registration form to obtain access to our app and their new wallet.

#### Implement Simple Authentication

Sometimes, user sign-up and sign-in features are the only requirement to consider when implementing user authentication. When complete, the app is able to talk to resources exposed to API Gateway or other cloud based services on authentication. Tyically, you can easily deploy a UserPool on Cognito by running Amplify’s addAuth command with the amplify-cli and using the default setup parameters. You can have your app retrieve the user’s authentication tokens to complete the authentication process by using either Auth.signUp and Auth.signIn.

#### AWS Authentication & Signing Requests

If you need to store videos or large files on AWS Simple Storage Service, your application will use services that will need signing requests authorized. Sending analytics or streaming data on Kinesis firehose to a data warehouse is another use-case to consider. Using credentials from a Cognito Identity Pool with a short *time to live*, and whose expiration and rotation are managed by AWS Amplify, all signs requests will automatically execute by calling Auth.signIn. All JWT Tokens from Cognito UserPools and your Cognito Identity Pools with return your user’s credentials which can always be accessed by the app using Auth.currentSession() and Auth.currentCredentials() as shown in the diagram below:

A close up of a logo

Description automatically generated

Using these tools we need to create a registration flow that will allow our users to:

1. Accept a user’s email and password to use for authentication and confirmation by Cognito.
2. Our app will obtain a user object from Amplify when the user is registered with Cognito.
3. A form is displayed to the user that will accept the emailed confirmation code as an input to be verfied against Cognito.
4. The app sends the code to Cognito to confirm the user’s registration and we complete the new user’s authentication.
5. We take the session from the object returned by Amplify on authentication and we update the application’s state.

First thing’s first; we must implement a proper registration form first!

“Your papers, please!” *Casablanca Policeman*A person sitting at a table

Description automatically generated

#### Implementing a Multi-Step User Registration React Component

The first thing we need to accomplish is to create a simple form that we can use to obtain the user’s initial registration information like email and password so that we can have Cognito send the user a confirmationCode that the app will use to confirm the user’s registration and authentication for the first time they log into the app.

There are a number of ways in which you can approach the registration of a user and the capturing of the data needed to authenticate a user. Many times you will want a responsive and multi-step registration workflow that will let your user enter multiple data points that your business will require you to persist to storage. In a multi-step workflow, the trick is to pass the values stored in the app’s state attributes to the child components that you will need to leverage to complete the registration workflow, as props that the final component can use to call the Amplify API when connecting to Cognito.

Since this application is a *FinTech App* it will more than likely need to abide by two key principles known within the financial industry as **KYC** and **AML** regulations; better known as *Know Your Customer* and *Anti-Money Laundering* laws. We will implement a *Multi-Step Registration Workflow* with a *Responsive UI/UX* that will be able to be displayed from any device.

We will create a multi-step registration component called UserRegistration that will pass down the appropriate state attributes to the child component that we will call UserConfirmation, which will use them to register, confirm, and authenticate our user against Amazon Cognito. We will also need to be able to access the userHasAuthenticated method in our props that we are using to update the application’s state from our parent compinent called src/App.js.

Moving forward with the Registration UI, we will touch upon a few more advanced topics pertaining to page *responsiveness* and laying out elements for optimized viewing on any device. *Mobile-First* is a design philosophy that every developer today must embrace or you risk losing out on large portions of market share. You must build all of your apps so that they can be viewed on any device and we can easily get this done with an npm library called react-media.

[react-media](https://www.npmjs.com/package/react-media) is a CSS media query component for React.

Every <Media> component listens for matches to a CSS media query and renders an element declared in your component based on whether the query matches or not.

Please continue on, and install this library at the root of your React client’s project directory with:

* $ npm install --save react-media

Generally, I find that the following sizes work best for most device sizes on iOS and Android platforms:

import React, { Component/\*, Fragment\*/ } from "react";  
// Using react containers to achieve responsiveness in Medias  
import Container from "react-bootstrap/Container";  
import Media from "react-media";  
  
<Container>  
 {/\* Smaller phones portrait \*/}  
 <Media query="(min-width: 319px) and (max-width: 567px)">  
 <p>KnowledgeBase Test 319 - 567</p>  
 </Media>  
  
 {/\* Smaller Landscapes \*/}  
 <Media query="(min-width: 568px) and (max-width: 639px)">  
 <p>KnowledgeBase Test 568 - 639</p>  
 </Media>  
  
 {/\* Bigger Landscapes \*/}  
 <Media query="(min-width: 640px) and (max-width: 767px)">  
 <p>KnowledgeBase Test 640 - 767</p>  
 </Media>  
  
 {/\* iPads \*/}  
 <Media query="(min-width: 768px) and (max-width: 991px)">  
 <p>KnowledgeBase Test 768 - 991</p>  
 </Media>  
  
 {/\* Desktops \*/}  
 <Media query="(min-width: 992px)">  
 <p>KnowledgeBase Test 992</p>  
 </Media>  
</Container>

You can find a [public gist on Github](https://gist.github.com/lopezdp/fc05d1bcfefd81e1f00d387f96f7d88c) where I have this information saved for you on the cloud **for-ev-er**! Do not hesitate to follow my profile and *star* my content!



alt text

With these fundamental concepts in mind, we now can proceed with the implementation of our multi-step registration workflow in React.js. Since we are building a wallet, we will ask a user for the following pieces of information:

* First Name
* Last Name
* Street Address
* City, State, ZipCode
* SSN (Government ID)
* Phone Number
* email
* password

We will keep this form simple and touch upon a few topics that will include passing the values of your state attributes as props down into a child component, passing a value as a function argument into a prop that will handle your application’s state in a parent component, and using state to render a new UI view while retaining the values in memory that you need to complete a user’s registration while keeping your app respinsive to all mobile devices.

We will start off by creating a few new components; remember that files in src/containers are React components that will make API calls to third party services or to our backend, while everything inside of src/components is just going to be UI components and views that we need in React.js to display our frontend logic appropriately. The first component we need is going to be the src/components/UserRegistration component that we will use to walk our users through the registration workflow we need to collect the data we need.

#### Implement UserRegistration and persist application state to Child components

Ok great, you have to implement a new component, you have to manage your application’s state, and somehow, you have to make sure that the correct state ends up in your Parent component once your userHasAuthenticated!

You have to remember, we created a function in our Parent component, src/App.js, that we called userHasAuthenticated which we have passed down as a prop into our Child components using our <Routes /> component. The function currently is a prop within the UserRegistration component because of the childProps that we declared in src/App.js with the following lines:

const childProps = {  
 isAuthenticated: this.state.isAuthenticated,  
 userHasAuthenticated: this.userHasAuthenticated  
 };

In order to change the state of our App, we need to make sure that the function that we are using to change the value of isAuthenticated is passed into the container that we will use within our registration component as a prop so that we can continue to manage our application’s state after a user logs into our App. Keep in mind that our homemade reducer is defined with the following implementation:

userHasAuthenticated = authenticated => {  
 this.setState({  
 isAuthenticated: authenticated  
 });  
 }

Shorlty, I will show you how we pass this prop into the nested component that we call from our UserRegistration component that I will now show you how to implement. Create a new file called src/components/UserRegistration.js and include the following source code that we will now review:

**<UserRegistration /> Component**

import React, { Component } from "react";  
import Container from "react-bootstrap/Container";  
import Media from "react-media";  
import Form from "react-bootstrap/Form";  
import ProgressBar from "react-bootstrap/ProgressBar";  
import { Auth } from "aws-amplify";  
import UiLoadBtn from "../components/UiLoadBtn";  
import config from "../config";  
import UserConfirmation from "../containers/UserConfirmation";  
import "./UserRegistration.css";  
  
export default class UserRegistration extends Component {  
  
 constructor(props) {  
 super(props);  
  
 this.state = {  
 isLoading: false,  
 govNumber: "",  
 streetAddress: "",  
 addressLine2: "",  
 firstName: "",  
 lastName: "",  
 mobilePhone: "",  
 email: "",  
 password: "",  
 confirmPassword: "",  
 confirmationCode: "",  
 validated: false,  
 registrationStep: 0,  
 newUser: null  
 };  
  
 // NOTE: Bind the registration workflow and the suggestion selector  
 this.registrationWorkFlow = this.registrationWorkFlow.bind(this);  
 this.validateForm = this.validateForm.bind(this);  
 }  
  
 //NOTE: When component mounts this takes over  
 componentDidMount() {  
 if (this.state.registrationStep.toString() === "0") {  
 this.setState({  
 registrationStep: 1  
 });  
 }  
 }  
  
 // FIXME: may need to delete or merge this into more  
 // efficient validation schemes  
 // Need to make sure the form is filled in with information  
 validateForm() {  
 let b = false;  
 // eslint-disable-next-line  
 console.log("validateForm called!!!!");  
  
 // If on step 1 && dealer has name && name is NaN && valid address  
 // then bool is true  
 if (this.state.registrationStep.toString() === "1") {  
 if (this.state.govNumber.length > 0 &&  
 isNaN(this.state.streetAddress) &&  
 this.state.streetAddress) {  
  
 b = true;  
 }  
 }  
  
 //validate fields in step 2 before submitting form  
 if (this.state.registrationStep.toString() === "2") {  
 if (this.state.firstName.length > 0 && isNaN(this.state.firstName) &&  
 this.state.lastName.length > 0 && isNaN(this.state.lastName) &&  
 this.state.mobilePhone.length === 10 && !isNaN(this.state.mobilePhone)) {  
 b = true;  
 }  
 }  
  
 //validate fields in step 3 before submitting form  
 if (this.state.registrationStep.toString() === "3") {  
 if (this.state.firstName.length > 0 &&  
 this.state.email.length > 0 &&  
 this.state.password.length > 0 &&  
 this.state.password === this.state.confirmPassword) {  
 // if false then confirmation is turned off!  
 b = true;  
 }  
 }  
  
 // return the bool value. true if form is valid or false if   
 // the form is invalid...  
 return (b);  
 }  
  
 handleChange = event => {  
 this.setState({  
 // using controlId change the state of the value as needed  
 [event.target.id]: event.target.value  
 });  
 }  
  
 handleSubmit = async event => {  
  
 const form = event.currentTarget;  
  
 // eslint-disable-next-line  
 console.log("This is the checkValidity res: " + form.checkValidity());  
  
 // NOTE: This will check the validity of the input  
 // fields with html5 validation tools  
 if (form.checkValidity() === false) {  
 event.preventDefault();  
 event.stopPropagation();  
 }  
  
 this.setState({  
 validated: true  
 });  
  
 event.preventDefault();  
  
 this.setState({  
 // set button spinner state to loading  
 // while we wait for async responses  
 isLoading: true  
 });  
  
 try {  
 // call cognito with amplify api  
 const newUser = await Auth.signUp({  
 username: this.state.email,  
 password: this.state.password  
 });  
  
 this.setState({  
 // set state with newUser data  
 newUser  
 });  
 } catch ( err ) {  
 // eslint-disable-next-line  
 alert(err.message);  
 }  
  
 this.setState({  
 // turn off loading when async complete  
 isLoading: false  
 });  
 }  
  
 renderProgressBar = (now) => {  
 const progressBar = <ProgressBar animated  
 variant="info"  
 now={ now }  
 label={ `${now}%` } />;  
 return (progressBar);  
 }  
  
 // TODO: Need functionality to handle state changes when button OnClick  
 registrationWorkFlow() {  
 this.setState({  
 isLoading: true  
 });  
  
 if (this.state.registrationStep.toString() === "1") {  
 this.setState({  
 registrationStep: 2  
 });  
 }  
  
 if (this.state.registrationStep.toString() === "2") {  
 // FIXME: Confirm Logic  
 this.setState({  
 registrationStep: 3  
 });  
 }  
  
 this.setState({  
 isLoading: false  
 });  
 }  
  
 renderStep1() {  
 //let registrationStep;  
  
 const { validated } = this.state.validated;  
  
 /\* if (this.state.registrationStep) {  
 registrationStep = this.state.registrationStep.toString();  
 } \*/  
  
 return (  
 <div>  
 <Media query="(min-width: 319px) and (max-width: 567px)">  
 <Container>  
 <header className="hdr1">  
 <h4 className="title">Register your Account</h4>  
 <h5 className="tagline">Step 1: Your basic information.</h5>  
 </header>  
  
 <section id="progressBar">  
 { this.renderProgressBar(5) }  
 </section>  
  
 <article className="signup">  
 { /\*Start Form for step 1 here!!\*/ }  
 <Form className="form"  
 noValidate  
 validated={ validated }  
 onSubmit={ this.handleSubmit }>  
 <p className="signup-tag">  
 <strong>Register to get access to your Wallet!</strong>  
 </p>  
   
 <Form.Group controlId="govNumber" className="gov-num">  
 <Form.Label>  
 SSN  
 </Form.Label>  
 <Form.Control type="text"  
 value={ this.state.govNumber }  
 onChange={ this.handleChange }  
 placeholder="444-77-3333"  
 required  
 minLength="9" />  
 <Form.Text className="help">  
 Your information is never shared!  
 </Form.Text>  
 </Form.Group>  
  
 { /\* NOTE: Implemented Google Geocode API to confirm zipCode & city by location \*/ }  
 <Form.Group controlId="streetAddress">  
 <Form.Label>  
 Street Address  
 </Form.Label>  
  
 <Form.Control type="text"  
 value={ this.state.streetAddress }  
 onChange={ this.handleChange }  
 placeholder="1234 Main St." />  
   
 <Form.Text className="help">  
 Please provide your street Address, and include your city, state, and zip code.  
 </Form.Text>  
 </Form.Group>  
  
 { /\* This is street Address \*/ }  
 <Form.Group controlId="addressLine2">  
 <Form.Label>  
 Suite or Unit  
 </Form.Label>  
 <Form.Control type="text"  
 value={ this.state.addressLine2 }  
 onChange={ this.handleChange }  
 placeholder="Ex: Unit #101" />  
 <Form.Text className="help">  
 Please provide a Suite or Unit number if available.  
 </Form.Text>  
 </Form.Group>  
  
 { /\* UiLoadBtn Component \*/ }  
 <UiLoadBtn block  
 onClick={ this.registrationWorkFlow }  
 size="lg"  
 disabled={ !this.validateForm() }  
 variant="primary"  
 className="nxtBtn"  
 isLoading={ this.state.isLoading }  
 text="Save Changes"  
 loadingText="Saving..."   
 />  
  
 </Form>  
 { /\* Section used for credibility seal on larger screens\*/ }  
 <section id="logo">  
 TBD - Section here  
 </section>  
 </article>  
 </Container>  
 </Media>  
  
 <Media query="(min-width: 568px)">  
 <Container>  
 <header className="hdr1">  
 <h4 className="title">Register your Account</h4>  
 <h5 className="tagline">Step 1: Basic Information about yourself.</h5>  
 </header>  
  
 <section id="progressBar">  
 { this.renderProgressBar(5) }  
 </section>  
  
 <article className="signup">  
 { /\*Start Form for step 1 here!!\*/ }  
 <Form className="form"  
 noValidate  
 validated={ validated }  
 onSubmit={ this.handleSubmit }>  
 <p>  
 <strong>Please Register and get access to your Wallet!</strong>  
 </p>  
   
 <Form.Group controlId="govNumber" className="gov-num">  
 <Form.Label>  
 SSN  
 </Form.Label>  
 <Form.Control type="text"  
 value={ this.state.govNumber }  
 onChange={ this.handleChange }  
 placeholder="444-77-3333"  
 required  
 minLength="9" />  
 <Form.Text className="help">  
 Your information is never shared!  
 </Form.Text>  
 </Form.Group>  
  
 <Form.Group controlId="streetAddress">  
 <Form.Label>  
 Street Address  
 </Form.Label>  
  
 <Form.Control type="text"  
 value={ this.state.streetAddress }  
 onChange={ this.handleChange }  
 placeholder="1234 Main St." />  
  
 <Form.Text className="help">  
 Please provide your street Address.  
 </Form.Text>  
 </Form.Group>  
  
 { /\* This is street Addreess \*/ }  
 <Form.Group controlId="addressLine2">  
 <Form.Label>  
 Suite or Unit  
 </Form.Label>  
 <Form.Control type="text"  
 value={ this.state.addressLine2 }  
 onChange={ this.handleChange }  
 placeholder="Ex: Unit #101" />  
 <Form.Text className="help">  
 Please provide a Suite or Unit number if available.  
 </Form.Text>  
 </Form.Group>  
  
 { /\* UiLoadBtn Component \*/ }  
 <UiLoadBtn block  
 onClick={ this.registrationWorkFlow }  
 size="lg"  
 disabled={ !this.validateForm() }  
 variant="primary"  
 className="nxtBtn"  
 isLoading={ this.state.isLoading }  
 text="Save Changes"  
 loadingText="Saving..."   
 />  
  
 </Form>  
 { /\* Section used for credibility seal on larger screens\*/ }  
 <section id="logo">  
 TBD - Section here  
 </section>  
 </article>  
 </Container>  
 </Media>  
 </div>  
 );  
 }  
  
 renderStep2(oType) {  
 //let registrationStep;  
  
 /\* if (this.state.registrationStep) {  
 registrationStep = this.state.registrationStep.toString();  
 } \*/  
  
 return (  
 <div>  
 <header className="hdr1">  
 <h4 className="title">Wallet Registration</h4>  
 <h5 className="tagline">Step 2: Create your User Account</h5>  
 </header>  
 <section id="progressBar">  
 { this.renderProgressBar(33) }  
 </section>  
 <div className="signup">  
 { /\*Start Form for step 1 here!!\*/ }  
 <Form className="form" onSubmit={ this.handleSubmit }>  
 <p>  
 <strong>Please Register As a New User</strong>  
 </p>  
   
 { /\* This is the user's first name and middle initial \*/ }  
 <Form.Group controlId="firstName">  
 <Form.Label>  
 First Name  
 </Form.Label>  
 <Form.Control type="text"  
 value={ this.state.firstName }  
 onChange={ this.handleChange }  
 placeholder="Ex: Henry J." />  
 <Form.Text className="help">  
 Enter your first name and an optional middle initial.  
 </Form.Text>  
 </Form.Group>  
 { /\* This is the user's last name and middle initial \*/ }  
 <Form.Group controlId="lastName">  
 <Form.Label>  
 Last Name  
 </Form.Label>  
 <Form.Control type="text"  
 value={ this.state.lastName }  
 onChange={ this.handleChange }  
 placeholder="Ex: Jones" />  
 <Form.Text className="help">  
 Enter your first name and an optional middle initial.  
 </Form.Text>  
 </Form.Group>  
 { /\* This is the user's mobile phone \*/ }  
 <Form.Group controlId="mobilePhone">  
 <Form.Label>  
 Mobile Phone  
 </Form.Label>  
 <Form.Control type="tel"  
 value={ this.state.mobilePhone }  
 onChange={ this.handleChange }  
 placeholder="Ex: (628) 425-2790" />  
 <Form.Text className="help">  
 Please <strong>ONLY</strong> include your 10-digit mobile phone number without any additional characters.  
 </Form.Text>  
 </Form.Group>  
 { /\* UiLoadBtn Component \*/ }  
 <UiLoadBtn block  
 onClick={ this.registrationWorkFlow }  
 size="lg"  
 disabled={ !this.validateForm() }  
 variant="primary"  
 className="nxtBtn"  
 isLoading={ this.state.isLoading }  
 text="Save Changes"  
 loadingText="Saving..." />  
 </Form>  
 { /\* Section used for credibility seal on larger screens\*/ }  
 <section id="logo">  
 TBD - Section here  
 </section>  
 </div>  
 </div>  
 );  
 }  
  
 renderStep3() {  
 // eslint-disable-next-line  
 let clsName;  
 // let registrationStep;  
  
 return (  
 <div>  
 <header className="hdr1">  
 <h4 className="title">Create a User Account</h4>  
 <h5 className="tagline">Step 3: Create your Login Credentials</h5>  
 </header>  
 <section id="progressBar">  
 { this.renderProgressBar(66) }  
 </section>  
 <div className="signup">  
 { /\*Start Form for step 1 here!!\*/ }  
 <Form className="form" onSubmit={ this.handleSubmit }>  
 <p>  
 <strong>Please Register As a new User.</strong>  
 </p>  
 <Form.Group controlId="email">  
 <Form.Label>  
 Email  
 </Form.Label>  
 <Form.Control autoFocus  
 size="lg"  
 type="email"  
 value={ this.state.email }  
 onChange={ this.handleChange }  
 required />  
 </Form.Group>  
 <Form.Group controlId="password">  
 <Form.Label>  
 Password  
 </Form.Label>  
 <Form.Control size="lg"  
 type="password"  
 value={ this.state.password }  
 onChange={ this.handleChange }  
 required />  
 </Form.Group>  
 <Form.Group controlId="confirmPassword">  
 <Form.Label>  
 Confirm Password  
 </Form.Label>  
 <Form.Control size="lg"  
 type="password"  
 value={ this.state.confirmPassword }  
 onChange={ this.handleChange } />  
 </Form.Group>  
 { /\* UiLoadBtn Component \*/ }  
 <UiLoadBtn block  
 size="lg"  
 type="submit"  
 disabled={ !this.validateForm() }  
 variant="primary"  
 className="nxtBtn"  
 isLoading={ this.state.isLoading }  
 text="Register"  
 loadingText="Processing..." />  
 </Form>  
 { /\* Section used for credibility seal on larger screens\*/ }  
 <section id="logo">  
 TBD - Section here  
 </section>  
 </div>  
 </div>  
 );  
 }  
  
 /\*  
 \*  
 \* @return <UserConfirmation />  
 \*  
 \* Reuturns the component that  
 \* will confirm a user's credentials  
 \*  
 \*/  
  
 renderConfirmationFormNew() {  
  
 return (  
 <UserConfirmation  
 registrationStep={ this.state.registrationStep }  
 email={ this.state.email }  
 password={ this.state.password }  
 isAuthenticated={ this.props.isAuthenticated }  
 userHasAuthenticated={ this.props.userHasAuthenticated }  
 user={ this.state } />   
 );  
 }  
  
 // NOTE: This is being used to get the correct workflow step  
 // to render. You have to setState outside of the render  
 // "canvas".  
 renderForms() {  
 let form;  
  
 if (this.state.registrationStep.toString() === "1") {  
 form = this.renderStep1();  
 } else if (this.state.registrationStep.toString() === "2") {  
 form = this.renderStep2(this.state.orgType);  
 } else if (this.state.registrationStep.toString() === "3") {  
 form = this.renderStep3();  
 }  
  
 return (form);  
  
 }  
  
 // TODO: Conditionally render each form element needed!!!  
 // conditionally render the registration form or the confirmation form  
 // depending on the user's registration state.  
 // ? this.renderForms()  
 render() {  
 // eslint-disable-next-line  
 console.log("THese are child props.isAuthenticated IN UR.js: " + this.props.isAuthenticated);  
 // eslint-disable-next-line  
 console.log("THese are child props.userHasAuthenticated IN UR.js: " + this.props.userHasAuthenticated);  
  
 return (  
 <div className="Registration">  
 { this.state.newUser === null  
 ? this.renderForms()  
 : this.renderConfirmationFormNew() }  
 </div>  
 );  
 }  
}

Carefully reviewing everything above, you may very well become easily overwhelmed, there is a lot going on and it may very well need to be reviewed many times before fully understanding everything going on. The first and most important thing we should discuss is the *explicit binding* of a few of our event handlers and *controlled components* inside of our constructor. The code in question looks like the following two lines:

* this.registrationWorkFlow = this.registrationWorkFlow.bind(this);
* this.validateForm = this.validateForm.bind(this);

In JavaScript, this binding really depends on how our functions are invoked; because of the way that React.js works with ES6 and **Class Components**, our event handlers and controlled components will point to an undefined value when our functions are used in strict mode. They will all fall back to their *default bindings* after losing their context. These functions must be implemented in our application’s constructor *explicitly* with a *hard bind* to the this value, while using the bind() method to persist the application context to the component rendered to the UI.

Please take some time to understand the reasons why this happens in JavaScript; binding this to our handlers and their respective instances inside of their constructors as shown above, lets us use our callback functions normally without losing their context.

Moving along, you should note that we use a state attribute that we are calling registrationStep to manage the step that the user is on within the registration workflow process. As soon as the components mounts, as you can see in the componentDidMount function, the state of the attribute changes to 1. You will find that this then tells React.js to use the registrationWorkFlow() function to render the first step in the multi-step registration form we have implemented for you.

We have split up the *views* that React.js will render on each step in the workflow based on the registrationStep, and each registrationStep will explicity tell React.js to render either renderStep1(), renderStep2(), or renderStep3() based on where the user is at in the process of registering their credentials with the app.

Finally, based on whether or not the newUser object has been created or not, the UI will render either the steps described above or the <UserConfirmation /> component. The <UserConfirmation /> is important because it is here where we need to pass in the props that we need it to use with out Parent component so that our app can update its state when a user isAuthenticated as we described above. As you can see below, we are using a *ternary* operation to tell React.js to display the correct component:

<div className="Registration">  
 { this.state.newUser === null  
 ? this.renderForms()  
 : this.renderConfirmationFormNew() }  
</div>

When the *ternary* operation fails and the conditional requires the new <UserConfimation /> component, react will call this.renderConfirmationFormNew() and our app will render the new component using the correct props as we declare below:

<UserConfirmation  
 registrationStep={ this.state.registrationStep }  
 email={ this.state.email }  
 password={ this.state.password }  
 isAuthenticated={ this.props.isAuthenticated }  
 userHasAuthenticated={ this.props.userHasAuthenticated }  
 user={ this.state }   
/>

If you have faithfully followed along, your new *multi-step registration* workflow in React.js should look like the image below:

*Multi-Step Registration UI*

A screenshot of a social media post

Description automatically generated

You have implemented a multi-step registration process while integrating a few of your services with React. In the next article we will discuss how to implement another feature that creates a database entry and we deploy all of these new features that take advantage of your GeneralLedger in the cloud on AWS Amplify.

### You completed the implementation React.js and a milti-step registration flow. Good Luck!

## Part 8: Implement your front end and deploy your app on AWS-Amplify.

* [Part 8: Integrate React with Serverless and Deploy on Amplify](https://github.com/lopezdp/TechnicalArticles/blob/master/DeployReactOnAmplify.md) - *Not Published.*