**Document Title**: Feature 1 Lambda Setup

**Project Title**: Hatching Sparrow CRM

**Date**: 6/30/2024

**Author(s)**: Ryan O’Connor, Full Stack Software Engineer

**Confidentiality:** N/A

**1. Backend Setup for Registration and Login**

**JWT Authentication System**:

* **User Registration**:
  + A user submits a registration form which is sent to a backend API endpoint.
  + The backend endpoint processes the registration data, stores the new user information in DynamoDB, and ensures that the user does not already exist.
  + Upon successful registration, a JWT token is created.
* **User Login**:
  + A user submits login credentials (username and password).
  + The backend API checks these credentials against the data stored in DynamoDB.
  + If the credentials are correct, the backend generates a JWT and sends it back to the client. This JWT will include claims such as the user ID, and any roles or permissions.
* **JWT Generation and Verification**:
  + Use a library such as jsonwebtoken in Node.js to generate and verify JWTs.
  + Configure a secret key to sign the JWTs, ensuring they cannot be tampered with.
  + Set an expiration time for each token to enhance security.

**AWS Lambda and API Gateway for Serverless API**:

* Set up AWS Lambda functions to handle the login and registration processes.
* Use API Gateway to expose these Lambda functions as HTTPS endpoints.

**Local Development Setup for Lambda Functions**

**1. Project Structure**: Set up your project on your local machine with the following structure:

/my-lambda-function

|-- node\_modules/

|-- package.json

|-- index.mjs

|-- lambda\_function\_payload.zip (generated for deployment)

**2. Initialize the Project**:

* Open a terminal in the directory where you want to create your Lambda function.
* Run npm init -y to create a package.json file.
* Install necessary packages, including aws-sdk if you’re using AWS services:

npm install aws-sdk

**3. Writing Your Lambda Function**:

* Create an index.mjs file (or .js if using CommonJS syntax) and implement your Lambda function logic as previously discussed.

**4. Local Testing**:

* While AWS Lambda does not run directly on your local machine without emulation, you can still execute and test the logic of your Lambda function. Use Node.js to run your script, simulating event inputs as needed.
* For a more accurate simulation, consider using tools like [AWS SAM (Serverless Application Model)](https://aws.amazon.com/serverless/sam/) or the [Serverless Framework](https://www.serverless.com/), which allow you to run Lambda functions in an environment that closely mimics AWS.

**Example of a Test File for Local Testing**

You can create a simple test script that invokes your Lambda function with a mock event object:

**test.js**:

import { handler as lambdaHandler } from './index.mjs';

const mockEvent = {

body: JSON.stringify({

username: "testuser",

password: "TestPassword123!",

email: "testuser@example.com"

})

};

lambdaHandler(mockEvent)

.then(response => console.log("Lambda Response:", response))

.catch(err => console.error("Error Invoking Lambda:", err));

Run this script using Node.js:

node test.js

**Packaging for Deployment**

**1. Ensure All Dependencies Are Included**:

* Your node\_modules directory should include all dependencies needed by your Lambda function.

**2. Create a ZIP File**:

* Zip the contents of your Lambda function directory, including index.mjs, node\_modules, and any other necessary files.

**3. Upload to Lambda**:

* Go to the AWS Lambda console and upload your ZIP file as the function code.

**Benefits of This Approach**

* **Development Flexibility**: You can use any IDE, access complete source control, and run npm commands locally.
* **Version Control**: Maintain your Lambda code in a Git repository, allowing for version tracking and collaboration.
* **Testing**: More robust testing by simulating the Lambda environment locally.
* **Dependency Management**: Easily manage npm packages without having to manually configure each in the AWS console.

By developing locally, you gain much more control over your Lambda function's development lifecycle, from coding and testing to deployment and maintenance. This approach is highly recommended for any non-trivial Lambda development.

**Frontend Setup for Register/Login (Simplified)**

1. **Create a React Application**:
   * Initialize the project using Create React App.
2. **Add Routing**:
   * Install React Router and set up basic routes for home, registration, and login pages.
3. **Develop Register and Login Components**:
   * Create forms in both components to handle user inputs and submissions.
   * Connect these forms to the respective backend API endpoints for user registration and login.

**Deployment on AWS**

1. **Build the React App**:
   * Run npm run build to generate the production build of your application.
2. **Configure S3 for Hosting**:
   * Create an S3 bucket and enable it for static website hosting.
   * Upload the build directory contents to the S3 bucket.
3. **Set Up CloudFront**:
   * Create a CloudFront distribution to serve your app from S3, enhancing load times with CDN caching.
4. **Optional DNS Configuration with Route 53**:
   * Use Route 53 for DNS management and point your domain to the CloudFront distribution.
5. **Environment Configuration**:
   * Manage API URLs and other environmental variables to ensure the frontend correctly interacts with the backend across different deployment stages.

**Step 1: Frontend Setup for Register/Login**

**1. Create a React Application**:

* If not already created, set up a new React application using Create React App:

npx create-react-app hatching-sparrow-frontend

cd hatching-sparrow-frontend

**2. Add Routing**:

* Install React Router for handling routing in your SPA:

npm install react-router-dom

* Set up basic routes in your application, including routes for home, registration, and login.

**Example Routing Setup**:

import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';

import Home from './components/Home';

import Register from './components/Register';

import Login from './components/Login';

function App() {

return (

<Router>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/register" element={<Register />} />

<Route path="/login" element={<Login />} />

</Routes>

</Router>

);

}

export default App;

**3. Build Register and Login Components**:

* **Register Component**: This component should include a form where users can enter their details (username, email, password), and it should handle form submissions by sending a POST request to your backend registration endpoint.
* **Login Component**: Similar to the register component, but the form will only require username and password. It will also post to your login endpoint.

**Example Form Handling in React**:

import React, { useState } from 'react';

import axios from 'axios';

function Register() {

const [userData, setUserData] = useState({ username: '', email: '', password: '' });

const handleChange = (e) => {

setUserData({ ...userData, [e.target.name]: e.target.value });

};

const handleSubmit = async (e) => {

e.preventDefault();

try {

const response = await axios.post('https://yourapi.com/register', userData);

console.log(response.data);

// handle redirect or storage of authentication tokens

} catch (error) {

console.error(error);

}

};

return (

<form onSubmit={handleSubmit}>

<input type="text" name="username" value={userData.username} onChange={handleChange} placeholder="Username" />

<input type="email" name="email" value={userData.email} onChange={handleChange} placeholder="Email" />

<input type="password" name="password" value={userData.password} onChange={handleChange} placeholder="Password" />

<button type="submit">Register</button>

</form>

);

}

export default Register;

**Step 2: Deploying the Frontend on AWS**

**1. Build Your React App**:

* Prepare your application for deployment:

npm run build

* This creates a build directory with a production build of your app.

**2. Hosting on AWS S3 and CloudFront**:

* **S3 Bucket Setup**:
  + Create an S3 bucket through the AWS Management Console.
  + Configure the bucket for static website hosting.
* **Upload Your Build**:
  + Upload the contents of the build directory to your S3 bucket.
* **CloudFront Distribution**:
  + Create a CloudFront distribution to serve your app from S3, which provides a CDN for faster global access and HTTPS support.
* **Route 53 for DNS (Optional)**:
  + Set up Route 53 to manage your domain’s DNS records and point your domain to your CloudFront distribution.

**3. Connect to the Backend**:

* Ensure that your frontend makes API calls to the backend hosted on AWS (Lambda via API Gateway). Use environment variables to manage API URLs to keep your frontend adaptable to different environments (development, staging, production).

By following these steps, you’ll have a fully functional frontend for your CRM system, capable of handling user registrations and logins, and securely deployed on AWS with a robust backend connection. This setup not only supports a scalable application architecture but also leverages AWS’s robust cloud infrastructure for optimal performance and reliability.

To connect your frontend application to your backend services, you typically use HTTP client libraries that facilitate sending requests to and receiving responses from a server. One of the most popular libraries for this purpose in the JavaScript ecosystem is **Axios**.

**What is Axios?**

Axios is a promise-based HTTP client for the browser and Node.js. It makes it easy to send asynchronous HTTP requests to REST endpoints and perform CRUD operations. It's widely favored due to its simple interface and comprehensive set of features.

**Key Features of Axios:**

1. **Make HTTP Requests**:
   * Supports all HTTP request methods like GET, POST, PUT, DELETE, etc.
2. **Transform Request and Response Data**:
   * Automatically transforms JSON data.
   * Allows modifying request and response data before they are handled by then or catch.
3. **Cancel Requests**:
   * Offers a way to cancel requests using cancel tokens.
4. **Interceptors**:
   * You can intercept requests or responses before they are handled by then or catch, allowing for logging, modification, or even stopping the request.
5. **Error Handling**:
   * Provides robust error handling capabilities through its promise-based architecture.
6. **Concurrent Requests**:
   * Allows handling multiple requests with methods like axios.all().

**Using Axios in a React Application**

Here’s how you might typically use Axios in a React project for making API calls:

**Installation**:

npm install axios

**Example Usage**:

import axios from 'axios';

function fetchData() {

axios.get('https://api.example.com/data')

.then(response => {

console.log('Data retrieved:', response.data);

})

.catch(error => {

console.error('Error fetching data:', error);

});

}

function submitData(userData) {

axios.post('https://api.example.com/register', userData)

.then(response => {

console.log('User registered:', response.data);

})

.catch(error => {

console.error('Error registering user:', error);

});

}

Given your current stack, which includes a React frontend and AWS services on the backend, **Axios** stands out as the best option for several reasons:

1. **Ease of Use**: Axios provides a straightforward and intuitive API for making HTTP requests, which can help streamline your code when interacting with backend services such as AWS Lambda functions exposed via API Gateway.
2. **Flexibility**: Axios allows you to easily configure requests to handle different requirements such as headers, query parameters, and timeout settings. This is particularly useful when dealing with AWS API Gateway, where you might need to pass custom headers for authentication tokens or API keys.
3. **Interceptors**: Axios interceptors are invaluable for handling repetitive tasks like injecting tokens into headers or handling errors globally. This is especially beneficial in applications that require authentication, as you can handle token renewal or redirection on authentication failures centrally.
4. **Compatibility**: Axios works seamlessly both in the browser and in Node.js environments, which provides flexibility in case parts of your system require server-side rendering or other Node.js-based functionalities.
5. **Community and Support**: Axios is widely used in the developer community, which means it has extensive documentation, a multitude of tutorials, and community support for troubleshooting. This can accelerate development and reduce downtime.
6. **Promise-Based**: Working with promises is essential for managing asynchronous logic in modern JavaScript applications, and Axios integrates well with async/await syntax in React, making your code cleaner and more readable.

**Integration Example**

To illustrate how Axios might integrate with your AWS backend through API Gateway:

* **Backend**: Your AWS Lambda functions handle business logic and communicate with other AWS services like DynamoDB.
* **API Gateway**: Set up as the HTTP endpoint for your Lambda functions, handling routing, authorization, and rate limiting.
* **Frontend**: Uses Axios to make HTTP requests to these endpoints, handling data display and user interactions in your React components.

**Conclusion**

Axios provides the robustness, ease of use, and extensive features that make it well-suited for your project’s needs. It aligns well with your AWS-powered backend and React frontend, ensuring you can handle complex scenarios like authentication, error handling, and state management efficiently.

If you are looking for a native alternative with no external dependencies, the **Fetch API** is a viable choice that is supported out of the box by modern browsers. It's slightly lower-level than Axios but is a standard part of the JavaScript runtime in browsers. However, for the best developer experience and feature set, Axios is recommended in your case.