Frontend - React Native App for Visualization of temperature

Created a React Native App which would call the following api’s –

1. http://ec2-52-10-248-119.us-west-2.compute.amazonaws.com:3000/getData

**The API will fetch the data from the sever for all the temperature’s and with its time-stamp**

The App component fetches temperature data at regular intervals and displays it in a line chart.

**State:**

1. timestamp: An array containing timestamps of the temperature data points.
2. temp: An array containing the temperature values.
3. loading: A boolean flag indicating whether data is still being fetched.

**Explanation:**

1. **Import Statements:**

The component imports necessary libraries like React, useState, useEffect, axios, Text, Dimensions, ActivityIndicator, and LineChart from 'react-native-chart-kit'.

1. **Functional Component:**

The component is defined as a functional component using the React.FC syntax (not shown in the provided code).

1. **State Variables:**

3.1) useState hook is used to manage three state variables:

3.2) timestamp: Initially set to an empty array to store timestamps.

3.3) temp: Initially set to an empty array to store temperature values.

3.4) loading: Initially set to true to indicate data is being fetched initially.

1. **useEffect Hook:**

4.1) The useEffect hook is used to fetch data at regular intervals:

4.1.1) An interval variable is created using setInterval to call a function every 5 seconds.

4.1.2) The function fetches data from the API endpoint http://ec2-52-10-248-119.us-west-2.compute.amazonaws.com:3000/getData using axios.get.

4.2) On successful response:

4.2.1) It extracts timestamps and temperatures from the response data.

4.2.2) It updates the timestamp and temp state variables by concatenating new data points with existing ones using the spread operator (...).

4.2.3) It checks if there are at least two data points and sets loading to false to hide the loading indicator.

4.3) On error:

It logs the error message to the console.

It sets loading to false to stop showing the loading indicator.

The cleanup function (return () => clearInterval(interval)) clears the interval when the component unmounts to prevent memory leaks.

1. **Chart Configuration (chartConfig):**

5.1) An object defining the visual style of the line chart using react-native-chart-kit properties.

5.2) It defines properties like background gradient colors, line color, stroke width, etc.

1. **Data Preparation (data):**

6.1) An object defining the data to be displayed in the chart.

6.2) It uses timestamp and temp state variables to create labels and datasets for the chart.

6.3) Labels are generated by converting the index of each data point to a string (e.g., "1", "2", etc.).

6.4) The dataset includes the temperature data points and a custom color for the line.

1. **Screen Width:**

7.1) It retrieves the screen width using Dimensions.get('window').width to adjust the chart width dynamically.

1. **Conditional Rendering:**

8.1) The component conditionally renders an ActivityIndicator while data is being fetched (loading is true).

8.2) Once data is fetched (loading is false), it renders the following elements:

A Text element displaying "Temperature-Time Chart".

A LineChart component from react-native-chart-kit with the following props:

data: The data object prepared earlier.

width: The width of the chart adjusted to fit the screen (minus 30px for padding).

height: The height of the chart set to 220px.

chartConfig: The chart configuration object.

1. **Export:**

9.1) The component is exported as the default export using export default App;.

This component demonstrates fetching data from an API at regular intervals, managing state, and using react-native-chart-kit to create a line chart for data visualization.

import React, { useState, useEffect } from 'react';

import { Text, Dimensions, ActivityIndicator } from 'react-native';

import axios from 'axios';

import { LineChart } from 'react-native-chart-kit';

const App = () => {

const [timestamp, setTimestamp] = useState([]);

const [temp, setTemp] = useState([]);

const [loading, setLoading] = useState(true);

useEffect(() => {

const interval = setInterval(() => {

axios.get('http://ec2-52-10-248-119.us-west-2.compute.amazonaws.com:3000/getData')

.then(response => {

const newTimestamps = response.data.map(entry => entry.timestamp);

const newTemps = response.data.map(entry => entry.temperature);

console.log("temp", newTemps);

console.log("time", newTimestamps);

setTimestamp(prevTimestamps => {

const updatedTimestamps = [...prevTimestamps, ...newTimestamps];

if (updatedTimestamps.length >= 2) {

setLoading(false);

}

return updatedTimestamps;

});

setTemp(prevTemps => {

const updatedTemps = [...prevTemps, ...newTemps];

if (updatedTemps.length >= 2) {

setLoading(false);

}

return updatedTemps;

});

})

.catch(error => {

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

setLoading(false); // Stop loading on error

});

}, 5000);

return () => clearInterval(interval);

}, []);

const chartConfig = {

backgroundGradientFrom: "#1E2923",

backgroundGradientFromOpacity: 0,

backgroundGradientTo: "#08130D",

backgroundGradientToOpacity: 0.5,

color: (opacity = 1) => `rgba(26, 255, 146, ${opacity})`,

strokeWidth: 2, // optional, default 3

barPercentage: 0.5,

useShadowColorFromDataset: false ,// optional

};

const data = {

labels: timestamp.map((ts, index) => (index + 1).toString()), // Converting index to string for labels

datasets: [

{

data: temp,

color: (opacity = 1) => `rgba(134, 65, 244, ${opacity})`, // optional

strokeWidth: 2 // optional

}

],

Legend:["Temperature"]

};

console.log("data.datasets = ", data.datasets[0].data);

const screenWidth = Dimensions.get('window').width;

if (loading) {

return <ActivityIndicator size="large" color="#0000ff" />;

}

return (

<>

<Text>Temperature-Time Chart (y-axis Temperature) </Text>

<LineChart

data={data}

width={screenWidth-16}

height={320}

chartConfig={chartConfig}

/>

<Text> x-axis Time</Text>

</>

);

};

export default App;

Backend Infra-

I have used free tier ec2 amazon aws instance linux machine.Added aws rds mysql database connection with the ec2 instance.

Deployed the Node js application using pm2.

[ec2-user@ip-172-31-21-87 wheatherServer]$ pm2 restart server1

Use --update-env to update environment variables

[PM2] Applying action restartProcessId on app [server1](ids: [ 0 ])

[PM2] [server1](0) ✓

┌────┬────────────┬─────────────┬─────────┬─────────┬──────────┬────────┬──────┬───────────┬──────────┬──────────┬──────────┬──────────┐

│ id │ name │ namespace │ version │ mode │ pid │ uptime │ ↺ │ status │ cpu │ mem │ user │ watching │

├────┼────────────┼─────────────┼─────────┼─────────┼──────────┼────────┼──────┼───────────┼──────────┼──────────┼──────────┼──────────┤

│ 0 │ server1 │ default │ 1.0.0 │ fork │ 120613 │ 0s │ 1 │ online │ 0% │ 19.2mb │ ec2-user │ disabled │

└────┴────────────┴─────────────┴─────────┴─────────┴──────────┴────────┴──────┴───────────┴──────────┴──────────┴──────────┴──────────┘

const express = require('express');

const bodyParser = require('body-parser');

const mysql = require('mysql');

const app = express();

app.use(bodyParser.json());

// MySQL connection configuration

const connection = mysql.createConnection({

host:host.env,

user:user.env, // replace with your database username

password:password.env, // replace with your database password

database: db.env // replace with your database name

});

// Connect to the database

connection.connect((err) => {

if (err) {

console.error('Error connecting to the database:', err);

return;

}

console.log('Connected to the MySQL database');

});

app.post('/data', (req, res) => {

const { timestamp, temperature, humidity, pressure } = req.body;

console.log("temperature = ",temperature);

console.log("timestamp = ",timestamp);

console.log("humidity = ",humidity);

console.log("pressure = ",pressure);

if (!timestamp || !temperature || !humidity || !pressure) {

return res.status(400).send('Invalid data format');

}

// Insert data into the MySQL database

const query = 'INSERT INTO SensorData (timestamp, temperature, humidity, pressure) VALUES (?, ?, ?, ?)';

connection.query(query, [timestamp, temperature, humidity, pressure], (err, results) => {

if (err) {

console.error('Unable to add data:', err);

return res.status(500).send('Error saving data');

}

res.status(200).send('Data saved successfully');

});

});

let offsetCounter = 0;

app.get('/getData', (req, res) => {

const query = 'SELECT \* FROM SensorData ORDER BY id ASC LIMIT 1 OFFSET ?'

connection.query(query,[offsetCounter] ,(err, results) => {

if (err) {

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

return res.status(500).send('Error fetching data');

}

console.log("getData results = ",results)

res.status(200).json(results);

offsetCounter++;

});

});

const port = process.env.PORT || 3000;

app.listen(port, () => {

console.log(`Server is running on port ${port}`);

});

**Application Description:**

This Express application acts as a server to handle data from a sensor and provides an API to access that data.

**Dependencies:**

* express: The web framework used to build the server.
* body-parser: A middleware used to parse incoming JSON data in requests.
* mysql: A driver used to interact with the MySQL database.

**Environment Variables:**

* host.env: Stores the hostname or IP address of the MySQL database server.
* user.env: Stores the username to access the MySQL database.
* password.env: Stores the password to access the MySQL database.
* db\_name.env: Stores the name of the database to connect to.

**Database Connection:**

* The application establishes a connection to the MySQL database using the provided credentials.
* It logs an error message if the connection fails and terminates the application.
* Upon successful connection, it logs a confirmation message.

**API Endpoints:**

1. **POST /data:**
   * This endpoint expects a JSON object in the request body containing the following properties:
     + timestamp: The timestamp of the sensor data.
     + temperature: The temperature value from the sensor.
     + humidity: The humidity value from the sensor.
     + pressure: The pressure value from the sensor.
   * It validates if all required properties are present in the request body.
   * If any property is missing, it responds with a 400 Bad Request error message.
   * Otherwise, it constructs a SQL query to insert the data into a table named SensorData in the database.
   * It executes the query using the connection.query method.
   * On successful insertion, it responds with a 200 OK message indicating successful data storage.
   * On error during insertion, it logs the error message and responds with a 500 Internal Server Error message.
2. **GET /getData:**
   * This endpoint retrieves data from the SensorData table in the database.
   * It uses a LIMIT clause to retrieve only one record at a time.
   * An OFFSET clause is used with a counter variable offsetCounter to retrieve subsequent records on subsequent calls. This simulates pagination-like behavior, fetching one record at a time with each request.
   * It executes a query to fetch data using the connection.query method.
   * On successful retrieval, it responds with a 200 OK message and the fetched data in JSON format.
   * On error during retrieval, it logs the error message and responds with a 500 Internal Server Error message.

**Server Startup:**

* The application listens for incoming connections on the specified port (3000 by default or the value set in the PORT environment variable).
* Upon successful startup, it logs a message indicating the port on which the server is running.

**Generation for temperature data**

**const axios = require('axios');**

**const { random } = require('lodash'); // To generate random numbers**

**const { DateTime } = require('luxon'); // To handle date and time**

**const url = "http://ec2-52-10-248-119.us-west-2.compute.amazonaws.com:3000/data";**

**const isoString = DateTime.utc().toISO();**

**const timeString = DateTime.fromISO(isoString).toFormat('HH.mm');**

**console.log(timeString);**

**// Function to generate random data**

**function generateData() {**

**return {**

**timestamp: timeString,**

**temperature: parseFloat((Math.random() \* (30.0 - 20.0) + 20.0).toFixed(2)),**

**humidity: parseFloat((Math.random() \* (70.0 - 40.0) + 40.0).toFixed(2)),**

**pressure: parseFloat((Math.random() \* (1020.0 - 1000.0) + 1000.0).toFixed(2))**

**};**

**}**

**// Function to send data to the server**

**async function sendData(url, data) {**

**try {**

**const response = await axios.post(url, data);**

**console.log(`Status: ${response.status}, Data: ${JSON.stringify(response.data)}`);**

**} catch (error) {**

**console.error(`Error sending data: ${error}`);**

**}**

**}**

**// Main function to generate and send data every 10 seconds**

**async function main() {**

**while (true) {**

**const data = generateData();**

**console.log("data = ", data);**

**await sendData(url, data);**

**await new Promise(resolve => setTimeout(resolve, 2000)); // Wait for 10 seconds**

**}**

**}**

**// Run the main function**

**main();**

**Libraries:**

* axios: Used to make HTTP requests (POST in this case).
* lodash: Used for the random function to generate random numbers.
* luxon: Used for the DateTime class to handle date and time manipulations.

**API Endpoint:**

* The script defines a constant url set to "<http://ec2-52-10-248-119.us-west-2.compute.amazonaws.com:3000/data>" which likely points to an API endpoint on a server that expects sensor data.

**Data Generation:**

* The generateData function creates a JavaScript object containing sensor data.
  + timestamp: It uses luxon to generate the current time in UTC format and then converts it to a human-readable format (e.g., "HH.mm").
  + temperature: It generates a random temperature value between 20.0 and 30.0 degrees Celsius (rounded to two decimal places).
  + humidity: It generates a random humidity value between 40.0 and 70.0 percent (rounded to two decimal places).
  + pressure: It generates a random pressure value between 1000.0 and 1020.0 hPa (rounded to two decimal places).

**Data Sending:**

* The sendData function is an asynchronous function that takes a URL and data object as arguments.
  + It uses axios.post to send a POST request to the specified URL with the provided data object in the request body.
  + On successful response:
    - It logs the response status code and the response data as a JSON string.
  + In case of an error:
    - It logs an error message.

**Main Function:**

* The main function is also asynchronous.
  + It creates an infinite loop (while(true)) to continuously generate and send data.
  + Inside the loop:
    - It calls generateData to create a new sensor data object.
    - It logs the generated data for debugging purposes.
    - It calls sendData to send the generated data to the API endpoint defined in url.
    - It uses await new Promise(resolve => setTimeout(resolve, 2000)) to wait for 2 seconds before the next iteration of the loop. This simulates sending data at regular intervals.

**Overall, this script demonstrates how to:**

* Generate random sensor data with timestamps.
* Send the generated data to a server API endpoint using an asynchronous HTTP request.
* Schedule data generation and sending at regular intervals.