CPS630 – Web Applications

Lab 2

Setting up React

Step 1: Install Node.js and npm

Before starting, ensure you have Node.js installed on your computer.

1. Visit https://nodejs.org to download and install Node.js. Node.js version 14.x or newer is required.
2. Download the latest version for your operating system (Windows, macOS, or Linux).
3. Follow the installation instructions provided on the website.
4. Node.js comes with npm (Node Package Manager), which you'll use to install Create React App and manage dependencies.

Step 2: Install Visual Studio Code

1. Go to the Visual Studio Code website.
2. Download the latest version for your operating system (Windows, macOS, or Linux).
3. Follow the installation instructions provided on the website.
4. After installation, open Visual Studio Code to ensure it is installed correctly.

Step 3: Install Create-React-App

Open a terminal or command prompt and install Create-React-App globally by running:

npm install -g create-react-app

This command allows you to use Create-React-App from anywhere on your system.

Step 4: Create a New React Project

To create a new React project, run:

create-react-app my-app

Replace 'my-app' with the desired project name. This command creates a new folder with your project name, sets up the initial project structure, and installs dependencies. In this lab, let us use “hello-world” as the project name.

Step 5: Open Project in VS Code

Navigate to your project directory using:

cd hello-world

Then, open the project in Visual Studio Code by running:

code .

This opens VS Code with your project folder.

Step 5: Start Development Server

To start the development server, run:

npm start

This command starts the development server and opens your default web browser to your new React app. The server will automatically reload if you make changes to the code.

Exercise 1: Creating a Simple React App with Components, Props, and Hooks

Step 1: Create a Functional Component

Modify the hello-world project as follows: Create a new file named `Greeting.js` in the `src` folder. Define a functional component that accepts props and returns a greeting message. For example:

```
function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}
export default Greeting;
```

Step 2: Use State with Hooks

In the `App.js` file, use the `useState` hook to manage state. For instance, create a state variable to hold a name and display it using the `Greeting` component:

```
import React, { useState } from 'react';
import Greeting from './Greeting';

function App() {
  const [name, setName] = useState('World');

  return (
    <div>
      <Greeting name={name} />
    </div>
  );
}

export default App;
```

Make sure you save ALL files. The server will automatically reload and display the message: Hello, World!

Exercise 2: Creating a Weather App

In this exercise, you will create an app that will fetch weather data from a public API and display it.

Step 1: Create a New React App

Create a new React application and call it: my-weather.

Step 2: Install Axios for API Requests

While in the project directory, install Axios to make HTTP requests by running:

npm install axios

Axios is a promise-based HTTP client for the browser and Node.js.

Step 3: Fetch Weather Data

Use the OpenWeatherMap API (or any other weather API) to fetch weather data. You'll need to sign up for an API key and use it in your requests.

Create a Component to Fetch Weather Data: In your `src` directory, create a new file named `Weather.js`. This component will handle the API call and display the weather data as follows:

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

const Weather = () => {
  const [weather, setWeather] = useState(null);
  const apiKey = 'YOUR_API_KEY'; // Replace with your OpenWeatherMap API key
  const city = 'Toronto'; // Example city

  useEffect(() => {
    const fetchWeather = async () => {
      try {
        const response = await
axios.get(`https://api.openweathermap.org/data/2.5/weather?q=${city}&appid=${apiKey}&u
nits=metric`);
        setWeather(response.data);
      } catch (error) {
```

```
            console.error("Error fetching weather data:", error);
        }
      };

      fetchWeather();
    }, [city]);

    if (!weather) return <div>Loading...</div>;

    return (
      <div>
        <h2>Weather in {city}</h2>
        <p>Temperature: {weather.main.temp}°C</p>
        <p>Condition: {weather.weather[0].main}</p>
      </div>
    );
};
```

Step 4: Display Weather Data

Update `App.js` to Include the Weather Component to display the weather data fetched from the API as follows:

```
import React from 'react';
import './App.css';
import Weather from './Weather';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <Weather />
      </header>
    </div>
  );
}

export default App;
```

Step 5: Run the App

Start the development server by running:

npm start

This will launch your React app in the default browser. You should now see the weather data displayed.

Step 6: input city name

Modify the Weather.js component so the user can input the city name. Hint: you'll need to add a state to handle the user input and a form to submit the city name.

Exercise 3: Creating a Blog App

In this exercise, we will create a simple blog platform where users can add posts. It incorporates props, state management with hooks, forms for user input, and routing for navigation between different components.

Step 1: Create the React App

Create a new React application called: my-blog

Step 2: Install React Router

Install React Router for navigation in your app directory:

npm install react-router-dom

Step 3: Add the following components

Create the following components in the `src` folder:
- `Home.js`: Displays the homepage.
- `Blog.js`: Displays blog posts.
- `NewPost.js`: Contains a form for adding a new post.

Home.js

The `Home.js` component displays a welcome message to the blog.

```
import React from 'react';

function Home() {
  return (
    <div>
      <h2>Welcome to the Blog!</h2>
      <p>This is a simple blog application built with React.</p>
    </div>
  );
}

export default Home;
```

Blog.js

The `Blog.js` component displays blog posts using state.

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

function Blog() {
  const [posts, setPosts] = useState([
    { id: 1, title: 'First Post', content: 'This is the first post.' },
    { id: 2, title: 'Second Post', content: 'This is the second post.' }
  ]);

  return (
    <div>
      <h2>Blog Posts</h2>
      {posts.map((post) => (
        <div key={post.id}>
          <h3>{post.title}</h3>
          <p>{post.content}</p>
        </div>
      ))}
    </div>
  );
}

export default Blog;
```

NewPost.js

The `NewPost.js` component contains a form for adding new blog posts.

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

function NewPost({ onAddPost }) {
  const [title, setTitle] = useState('');
  const [content, setContent] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    const newPost = { id: Date.now(), title, content };
    onAddPost(newPost); // This function would be passed down from a parent component or
```

```
context
  setTitle('');
  setContent('');
 };

 return (
  <div>
   <h2>Add a New Post</h2>
   <form onSubmit={handleSubmit}>
    <div>
     <label>Title:</label>
     <input type="text" value={title} onChange={(e) => setTitle(e.target.value)} />
    </div>
    <div>
     <label>Content:</label>
     <textarea value={content} onChange={(e) => setContent(e.target.value)}></textarea>
    </div>
    <button type="submit">Add Post</button>
   </form>
  </div>
 );
}

export default NewPost;
```

Step 4: Implement Routing

In `App.js`, use `BrowserRouter` and `Route` components from `react-router-dom` to set up routes for your components. For example, route `/` to `Home`, `/blog` to `Blog`, and `/new-post` to `NewPost`.

Your app should look something like this:

Exercise 4: Integrating Node.js and Express with my-blog App

In this exercise, you will extend my-blog to include a backend server using Node.js and Express. The server will handle API requests for adding and fetching blog posts.

Step 1: Set Up the Backend Server

Create a file named `server.js` in your project root. This file will set up your Express server.

Install Express by running `npm install express` in your root directory.

Step 2: Implement the Server

In `server.js`, set up a basic Express server:

```
const express = require('express');
const cors = require('cors'); // CORS is a node.js package for providing a Connect/Express
// middleware that can be used to enable CORS with various options. Make sure it is installed.
const app = express();
const PORT = process.env.PORT || 3001;

// Middleware to parse JSON requests
app.use(express.json());

// Enable CORS for all origins
app.use(cors());

// Example posts array (initial data)
// TO DO: Initialize your blog posts data

// Route to get all blog posts
app.get('/api/posts', (req, res) => {
  res.json(posts);
});

app.get('/api/homeData', (req, res) => {
   const homeData = {
     message: 'Welcome to our website! Explore our blog for interesting articles.'
   };
   res.json(homeData);
 });

// Route to create a new blog post
app.post('/api/posts', (req, res) => {
  const { title, content } = req.body;
```

```
  if (!title || !content) {
    return res.status(400).json({ error: 'Title and content are required' });
  }

  const newPost = { id: Date.now(), title, content };
  posts.push(newPost);
  res.status(201).json(newPost);
});

// Start the server
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

To do: Update the React App to Use the Backend

Modify the `Blog` and `NewPost` components to fetch and add posts from/to the backend server using `fetch`. Note: `NewPost` component simply displays a message. It does not modify the content of the `server.js`. See screenshots below.

Step 3: Run the Backend Server

Navigate to the `server` folder and start the Express server by running:

node server.js

Your server will now be running on `http://localhost:3001`.

Step 4: Running the Full App

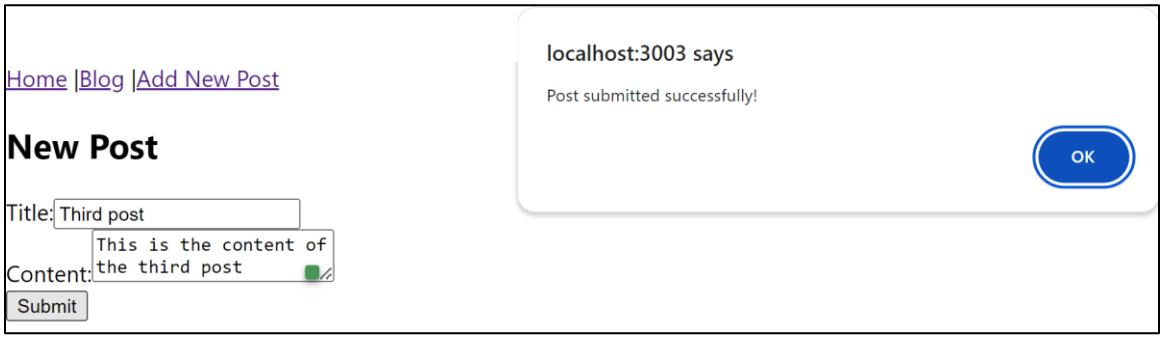
With the backend server running, start the React frontend in development mode by running `npm start` in the project root. The React app will now communicate with your Express backend for adding and fetching blog posts.

Your app should look something like this:

Home page

Blog page

New Post page



Exercise 5: Integrating my-blog App with MongoDB for Data Storage

In this exercise, you will extend my-blog to store the blog data in a MongoDB database.

Step 1: Install MongoDB server on your local machine

Here are the general steps you need to follow:

1. Install MongoDB Locally: Download and install MongoDB Community Edition from the official MongoDB website. Follow the installation instructions for Windows, including setting up the data directory and environment variables if necessary.

2. Start the MongoDB Server: After installation, start the MongoDB server. You can start MongoDB with the `mongod` command if you've added MongoDB to your system's PATH, or by navigating to the MongoDB `bin` directory and running `mongod` from there.

Visit https://www.mongodb.com/docs/manual/installation/ for more details.

Step 2: Install Mongoose

Installing Mongoose with `npm install mongoose` in your Node.js project allows you to interact with MongoDB through your application code. However, Mongoose acts as a bridge between your Node.js application and MongoDB; it does not install MongoDB itself. To work with MongoDB data, you still need a MongoDB server running either locally on your machine or hosted on a cloud platform (like MongoDB Atlas).

Step 3: Connect Mongoose to MongoDB

In your `server.js`, use Mongoose to connect to the MongoDB server with a connection string. If you're running MongoDB with default settings locally, your connection string would be something like `mongodb://localhost:27017/yourDatabaseName`.

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/blogDB', { useNewUrlParser: true, useUnifiedTopology: true });
```

Step 4: Define Schema and Model

```
const postSchema = new mongoose.Schema({ title: String, content: String });
const Post = mongoose.model('Post', postSchema);
```

Step 5: Modify GET Route

```
app.get('/api/posts', async (req, res) => {
 try {
  const posts = await Post.find({});
  res.json(posts);
 } catch (err) {
  res.status(500).send(err);
 }
});
```

Step 6: Modify POST Route:

Modify your POST route accordingly.

Step 7: Remove Static Posts Array

Make sure you delete the static posts array. Now, your data will be stored in the database.

Step 8: Update the React App

Modify the `Blog` and `NewPost` components (if needed)

Step 9: Start Your Server and Launch your App

Your MongoDB server must be running. Start your express server and launch your App as you did in Exercise 4.
Your app should look something like this:

New Post page

Blog page

Deliverables:

1. Create a folder called *YourLastName_YourFirstName_YourStudent#_Lab2*
2. Inside this folder, create five folders: *Exercise1*, *Exercise2*, etc.
3. In each Exercise folder, put the corresponding source code and a video demo in .mp4 format.
    a. Save the Lab2 folder in your TMU Google Drive and share (make sure you provide read permission to TMU members)
    b. Zip the Lab2 folder and upload it to D2L. Provide us with a link to your shared Google Drive in the comment section.
4. **Your submission must be complete according to the guidelines above. Failure to do so will result in a mark of Zero on the lab.**

Evaluation Criteria:

1. Functionality: Does the code work as expected without bugs?
2. Code Quality: Is the code well-organized, commented, and following best practices?
3. Design: How visually appealing and user-friendly is the interface?

Submission Deadline:

4. **Tuesday, March 12, 2024 @ 11:59 pm**