National Forensic Sciences University
Delhi Campus
(An Institute of National Importance)

Advance Web Development

Practical File

Subject Code: CTMTCS SVI P6 EL2

Submitted to
Dr. Parul Arora

Submitted by
Rupam Barui
102CTMBCS2122002

Contents

Experiment 1: Build a single page application using React. 2
Program . . . 2
Output . . . 5

Experiment 2: Create custom HTML elements using the web components standard 6
Program . . . 7
Output . . . 9

Experiment 3: Build a REST API using Node.js 10
Program . . . 11
Output . . . 12

Experiment 4: Implement CRUD operations and integrate a database 13
Program . . . 14
Output . . . 18

Experiment 5: Set up a GraphQL Server 19
Program . . . 19
Output . . . 22

Experiment 6: To create React Component 24
Program . . . 24
Output . . . 27

Experiment 7: Implement State and Props in React 28
Program . . . 28
Output . . . 30

Experiment 8: Implement Hooks (useState and useEffect) in React 31
Program . . . 31
Output . . . 33

Experiment 9: How to handle form elements in React 34
Program . . . 34
Output . . . 38

Experiment 10: Implement Node.js events 39
Program . . . 39
Output . . . 41

Experiment 11: Pass UserID to Frontend with Firebase 42
Program . . . 42
Output . . . 48

Experiment No. 1: Build a single page application using React.

Objective

The objective of this experiment is to implement a simple React Single Page Application (SPA) for a Todo app that displays predefined tasks and stores their completion state in local storage.

Introduction

In this experiment, we will implement a Todo app using React. The app will display a list of predefined tasks and allow users to mark them as completed. The completion state of each task will be stored in the browser's local storage to persist across sessions.

Theory

React is a popular JavaScript library for building user interfaces, especially single-page applications. In this experiment, we will use React's functional components and hooks (‘useState‘ and ‘useEffect‘) to manage the state of the tasks and interact with local storage.

Algorithm

1. Define the initial tasks as an array of objects, each with a ‘text‘ and ‘completed‘ property.
2. Create a React component that initializes the state of tasks from local storage.
3. Use the ‘useEffect‘ hook to update local storage whenever the state of tasks changes.
4. Implement a function to toggle the completion state of a task.
5. Render the tasks as a list, with each task displayed as an item that can be marked as completed.

Program

```
// App.js
import React, { useState, useEffect } from 'react';

// Initial tasks provided by the user
const initialTasks = [
  "Build a single page application using React.",
  "Create custom HTML elements using the web components standard.",
  "Build a rest-API using Node.js.",
  "Implement CRUD operations and integrate a database/mongoDB.",
  "Set up a Graph QL server.",
  "To create React Component.",
  "Implement State and Props in React.",
  "Implement Hooks (useState and useEffect) in React.",
```

```
  "How to handle form elements in React.",
  "Implement node.js events.",
  "Create database in MySQL and implement operations on it.",
  "Pass UserID to frontend."
];

// Utility functions for local storage
const getLocalStorage = () => {
  const data = localStorage.getItem('tasks');
  return data ? JSON.parse(data) : initialTasks.map(task => ({ text: task,
      completed: false }));
};

const setLocalStorage = (tasks) => {
  localStorage.setItem('tasks', JSON.stringify(tasks));
};

const App = () => {
  const [tasks, setTasks] = useState(getLocalStorage());

  useEffect(() => {
    setLocalStorage(tasks);
  }, [tasks]);

  const toggleTaskCompletion = (index) => {
    const updatedTasks = tasks.map((task, i) =>
      i === index ? { ...task, completed: !task.completed } : task
    );
    setTasks(updatedTasks);
  };

  return (
    <div className="min-h-screen bg-gray-100 flex items-center justify-center
        p-6">
      <div className="w-full max-w-md bg-white rounded-lg shadow-lg p-6">
        <h1 className="text-2xl font-bold mb-4">Todo App</h1>
        <ul className="space-y-2">
          {tasks.map((task, index) => (
            <li
              key={index}
              className={`p-2 border border-gray-300 rounded flex
                  justify-between items-center ${task.completed ? 'line-through
                  text-gray-500' : ''
                }`}
            >
```

```
            <span onClick={() => toggleTaskCompletion(index)}
                classNameName="cursor-pointer flex-1">{task.text}</span>
          </li>
        ))}
      </ul>
    </div>
  </div>
 );
};

export default App;
```

Conclusion

In this experiment, we successfully implemented a simple React Single Page Application for a Todo app that displays predefined tasks and stores their completion state in local storage.

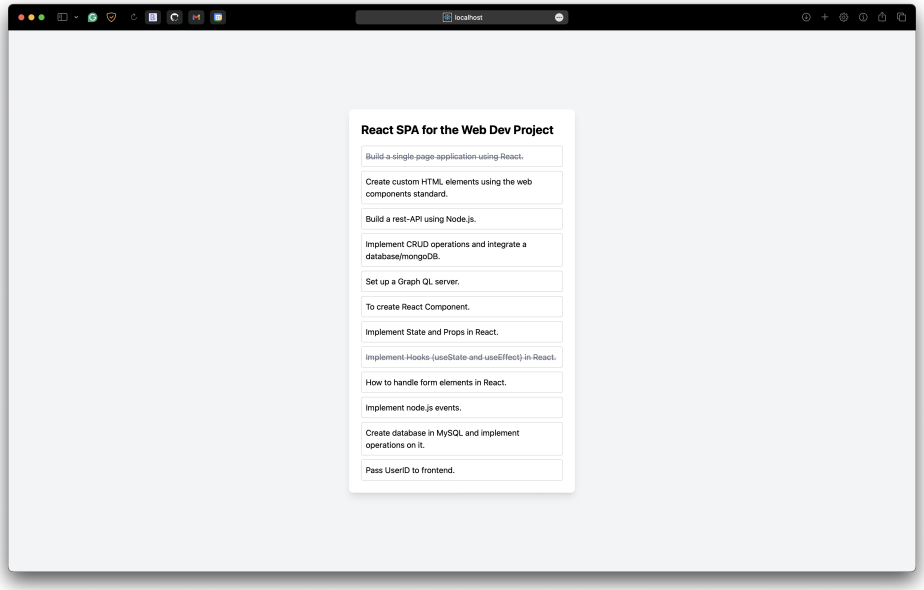

Figure 1: SPA Frontend

Experiment No. 2: Create custom HTML elements using the web components standard

Objective

The objective of this experiment is to create custom HTML elements using the Web Components standard, which includes the use of custom elements, shadow DOM, and HTML templates.

Introduction

Web Components is a suite of different technologies allowing you to create reusable custom elements with their functionality encapsulated away from the rest of your code. In this experiment, we will create a custom web component that displays a simple message.

Theory

The Web Components standard consists of four main technologies:

- Custom Elements: APIs to define new HTML elements.
- Shadow DOM: APIs to encapsulate the internal structure of the element.
- HTML Templates: `<template>` and `<slot>` elements for defining templates.
- HTML Imports (deprecated): Mechanism for loading and using HTML documents in other HTML documents.

In this experiment, we will use Custom Elements and Shadow DOM.

Algorithm

1. Define a class that extends ‘HTMLElement‘.
2. Create a shadow root for the custom element.
3. Attach a template to the shadow root.
4. Define the custom element using ‘customElements.define‘.
5. Add the custom element to the HTML page.

Program

```
class MyCustomElement extends HTMLElement {
   constructor() {
      super();

      const shadow = this.attachShadow({ mode: "open" });

      const template = document.createElement("template");
      template.innerHTML = `
            <style>
                :host {
                    display: block;
                    margin-bottom: 20px;
                    border-radius: 8px;
                    overflow: hidden;
                    box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
                }

                .container {
                    font-family: 'Arial', sans-serif;
                    padding: 20px;
                    background-color: #ffffff;
                    border: 1px solid #e0e0e0;
                    border-radius: 8px;
                }
            </style>
            <div class="container">
                <slot></slot>
            </div>
        `;

      shadow.appendChild(template.content.cloneNode(true));
   }
}

customElements.define("my-custom-element", MyCustomElement);
```

HTML Usage

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <meta charset="UTF-8" />
```

```
      <meta
        name="viewport"
        content="width=device-width, initial-scale=1.0"
      />
      <title>Custom Element Example</title>
      <link
        rel="stylesheet"
        href="styles.css"
      />
    </head>
    <body>
      <h1 class="headingC">Custom Element Example</h1>
      <my-custom-element>
        Hello, this is a custom element!
      </my-custom-element>

      <script src="custom-element.js"></script>
    </body>
</html>
```

Conclusion

In this experiment, we successfully created a custom HTML element using the Web Components standard. We defined a custom element, attached a shadow DOM, and used a template to encapsulate the element's structure and style.

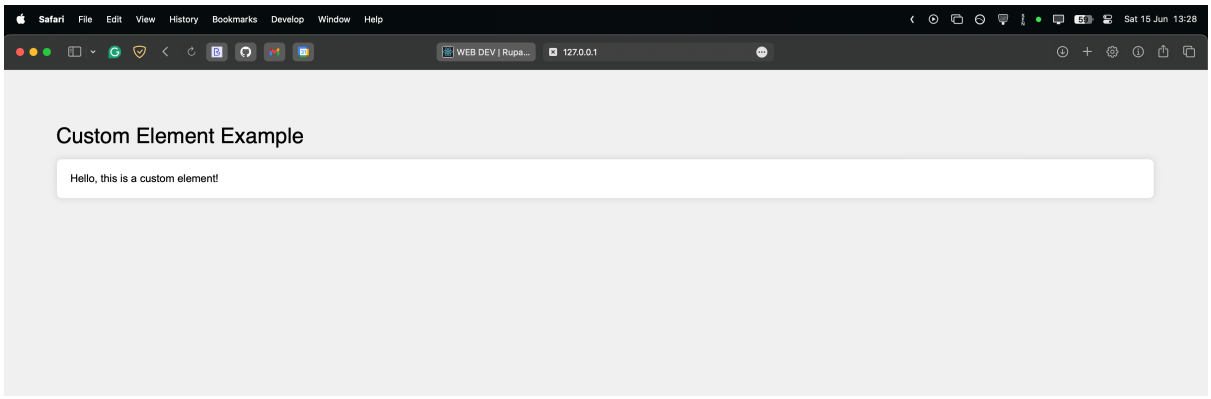

Figure 2: Custom Component

Experiment No. 3: Build a REST API using Node.js

Objective

The objective of this experiment is to build a simple REST API using Node.js that returns the day of the week for a given date.

Introduction

A RESTful API (Representational State Transfer) is an architectural style for designing networked applications. Node.js, with its non-blocking event-driven architecture, is well-suited for building scalable APIs. In this experiment, we will create a REST API that, given a date, returns the corresponding day of the week.

Theory

In this experiment, we will:

- Use Node.js to create an HTTP server.
- Define a route for handling HTTP GET requests.
- Parse date parameters from the query string.
- Use the ‘date-fns‘ library to determine the day of the week.
- Return the result as a JSON response.

Algorithm

1. Set up a new Node.js project.
2. Install necessary packages like ‘express‘ for routing and ‘date-fns‘ for date formatting.
3. Define a route (‘GET /day-of-week‘) that accepts ‘day‘, ‘month‘, and ‘year‘ as query parameters.
4. Parse the date parameters and construct a JavaScript ‘Date‘ object.
5. Use ‘date-fns‘ to determine the day of the week.
6. Return the day of the week as a JSON response.
7. Test the API endpoint using tools like Postman or ‘curl‘.

Program

```
// app.js
const express = require("express");
const { format } = require("date-fns");

const app = express();
const port = 6969;

app.get("/day-of-week", (req, res) => {
   const { day, month, year } = req.query;

   try {
      // Construct the date object
      const date = new Date(year, month - 1, day); // month - 1 because months
          are 0-indexed in JavaScript

      // Check if the date is valid
      if (isNaN(date)) {
         throw new Error("Invalid date format.");
      }

      // Get the day of the week
      const dayOfWeek = format(date, "EEEE");

      // Send the response with the day of the week
      res.json({ dayOfWeek });
   } catch (error) {
      // Handle errors, e.g., invalid date format
      res.status(400).json({ error: error.message });
   }
});

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

Conclusion

In this experiment, we successfully built a REST API using Node.js. The API accepts a date in the form of day, month, and year as query parameters, and returns the corresponding day of the week. This demonstrates the basic principles of creating a RESTful service with Node.js.

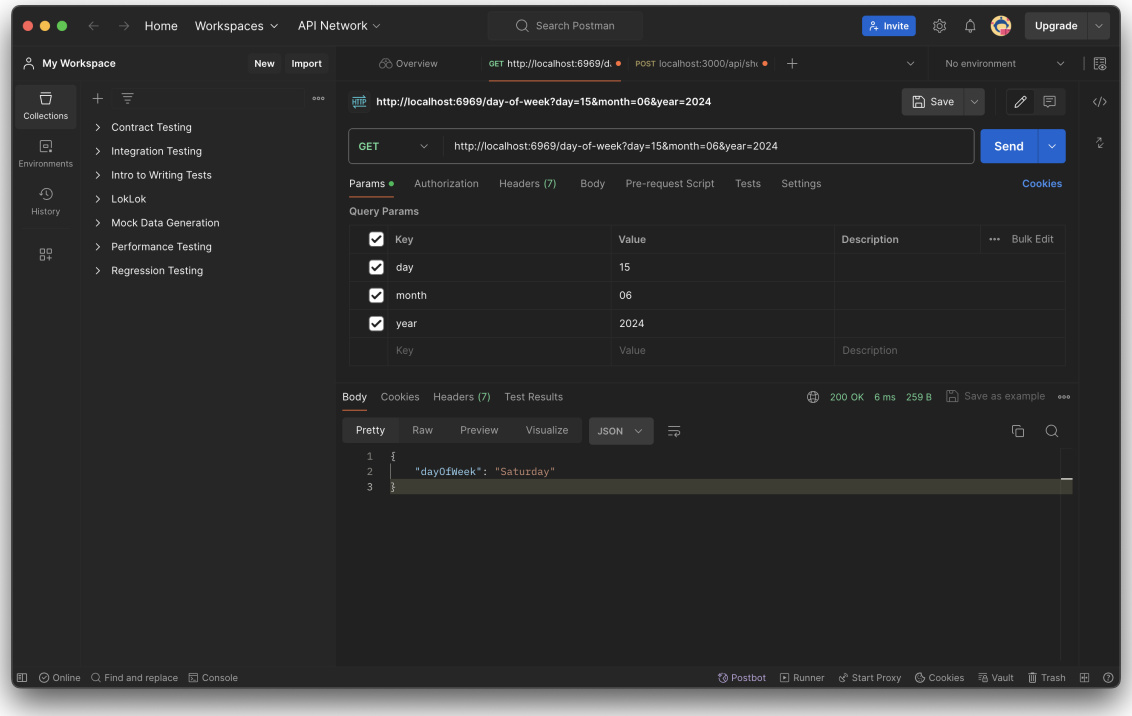

Figure 3: Output of the REST API showing the day of the week for a given date

Experiment No. 4: Implement CRUD operations and integrate a database

Objective

The objective of this experiment is to build a simple ToDo app using React and Firebase. The app will include CRUD (Create, Read, Update, Delete) operations and user authentication via Google Login.

Introduction

Firebase provides a suite of cloud-based services including a real-time NoSQL database, user authentication, and hosting. In this experiment, we will use Firebase Firestore for storing and managing tasks and Firebase Authentication for user login. The app will allow users to add, edit, delete, and view tasks. Each user will only see their own tasks.

Theory

In this experiment, we will:

- Set up Firebase and integrate it with a React application.
- Use Firebase Authentication to enable Google Login.
- Use Firestore to perform CRUD operations on tasks.
- Display a personalized greeting message and a list of tasks for the logged-in user.

Algorithm

1. Set up a new React project using Create React App.
2. Install necessary packages: ‘firebase‘, ‘react-firebase-hooks‘, and ‘tailwindcss‘.
3. Initialize Firebase with the project’s configuration.
4. Implement user authentication using Google Login.
5. Define Firestore queries to fetch, add, update, and delete tasks.
6. Display the tasks in a user-friendly interface using Tailwind CSS.
7. Test the app to ensure all functionalities work correctly.

Program

```
// App.js
import { initializeApp } from "firebase/app";
import {
  GoogleAuthProvider,
  getAuth,
  signInWithPopup,
  signOut,
} from "firebase/auth";
import {
  addDoc,
  collection,
  deleteDoc,
  doc,
  getFirestore,
  onSnapshot,
  query,
  where,
  updateDoc,
} from "firebase/firestore";
import React, { useEffect, useState } from "react";
import { useAuthState } from "react-firebase-hooks/auth";

const firebaseConfig = {
  apiKey: "AIzaSyCiY_lVqqd9JG1WifeLANcH1u5KnHdlm8M",
  authDomain: "web-dev-project-sem6.firebaseapp.com",
  projectId: "web-dev-project-sem6",
  storageBucket: "web-dev-project-sem6.appspot.com",
  messagingSenderId: "228672974177",
  appId: "1:228672974177:web:00bf245faa0101b253e920",
};

initializeApp(firebaseConfig);
const auth = getAuth();
const provider = new GoogleAuthProvider();
const db = getFirestore();

const App = () => {
  const [user] = useAuthState(auth);
  const [tasks, setTasks] = useState([]);
  const [newTask, setNewTask] = useState("");
  const [isEditing, setIsEditing] = useState(false);
  const [currentTaskId, setCurrentTaskId] = useState(null);

  useEffect(() => {
```

```
  if (user) {
    const q = query(collection(db, "tasks"), where("userId", "==",
        user.uid));
    const unsubscribe = onSnapshot(q, (querySnapshot) => {
      const tasksData = [];
      querySnapshot.forEach((doc) => {
        tasksData.push({ ...doc.data(), id: doc.id });
      });
      setTasks(tasksData);
    });
    return () => unsubscribe();
  }
}, [user]);

const handleLogin = async () => {
  await signInWithPopup(auth, provider);
};

const handleLogout = () => {
  signOut(auth);
};

const addTask = async () => {
  if (newTask.trim() === "") return;
  await addDoc(collection(db, "tasks"), {
    text: newTask,
    createdAt: new Date(),
    userId: user.uid,
  });
  setNewTask("");
};

const editTask = (task) => {
  setIsEditing(true);
  setNewTask(task.text);
  setCurrentTaskId(task.id);
};

const updateTask = async () => {
  if (newTask.trim() === "") return;
  const taskDoc = doc(db, "tasks", currentTaskId);
  await updateDoc(taskDoc, { text: newTask });
  setNewTask("");
  setIsEditing(false);
  setCurrentTaskId(null);
};
```

```
const deleteTask = async (id) => {
  await deleteDoc(doc(db, "tasks", id));
};

return (
  <div className="min-h-screen bg-gradient-to-r from-green-200 to-blue-200
      flex flex-col items-center justify-center p-4">
    <div className="bg-white p-6 rounded-lg shadow-lg w-full max-w-md">
      <h1 className="text-3xl font-bold mb-6 text-center">ToDo App</h1>
      {user ? (
        <>
          <p className="text-xl mb-4">
            Hello, {user.displayName}! These are your tasks:
          </p>
          <button
            onClick={handleLogout}
            className="bg-red-500 text-white px-4 py-2 rounded
                hover:bg-red-600 mb-4 w-full"
          >
            Logout
          </button>
          <div className="mb-4">
            <input
              type="text"
              className="border p-2 w-full rounded mb-2"
              placeholder="Enter task"
              value={newTask}
              onChange={(e) => setNewTask(e.target.value)}
            />
            <button
              onClick={isEditing ? updateTask : addTask}
              className={`${
                isEditing ? "bg-yellow-500 hover:bg-yellow-600" :
                    "bg-blue-500 hover:bg-blue-600"
              } text-white px-4 py-2 rounded w-full`}
            >
              {isEditing ? "Update Task" : "Add Task"}
            </button>
          </div>
          <ul>
            {tasks.map((task) => (
              <li
                key={task.id}
                className="flex justify-between items-center bg-gray-100 p-3
                    my-2 rounded shadow-sm"
```

```
              >
                <span>{task.text}</span>
                <div>
                  <button
                    onClick={() => editTask(task)}
                    className="bg-yellow-500 text-white px-4 py-2 rounded
                        hover:bg-yellow-600 mr-2"
                  >
                    Edit
                  </button>
                  <button
                    onClick={() => deleteTask(task.id)}
                    className="bg-red-500 text-white px-4 py-2 rounded
                        hover:bg-red-600"
                  >
                    Delete
                  </button>
                </div>
              </li>
            ))}
          </ul>
        </>
      ) : (
        <button
          onClick={handleLogin}
          className="bg-green-500 text-white px-4 py-2 rounded
              hover:bg-green-600 w-full"
        >
          Login with Google
        </button>
      )}
    </div>
  </div>
 );
};

export default App;
```

Conclusion

In this experiment, we successfully built a ToDo application using React and Firebase. The app implements CRUD operations and integrates user authentication with Google Login. Each user can manage their tasks, demonstrating the use of Firebase Firestore and Authentication in a React project.

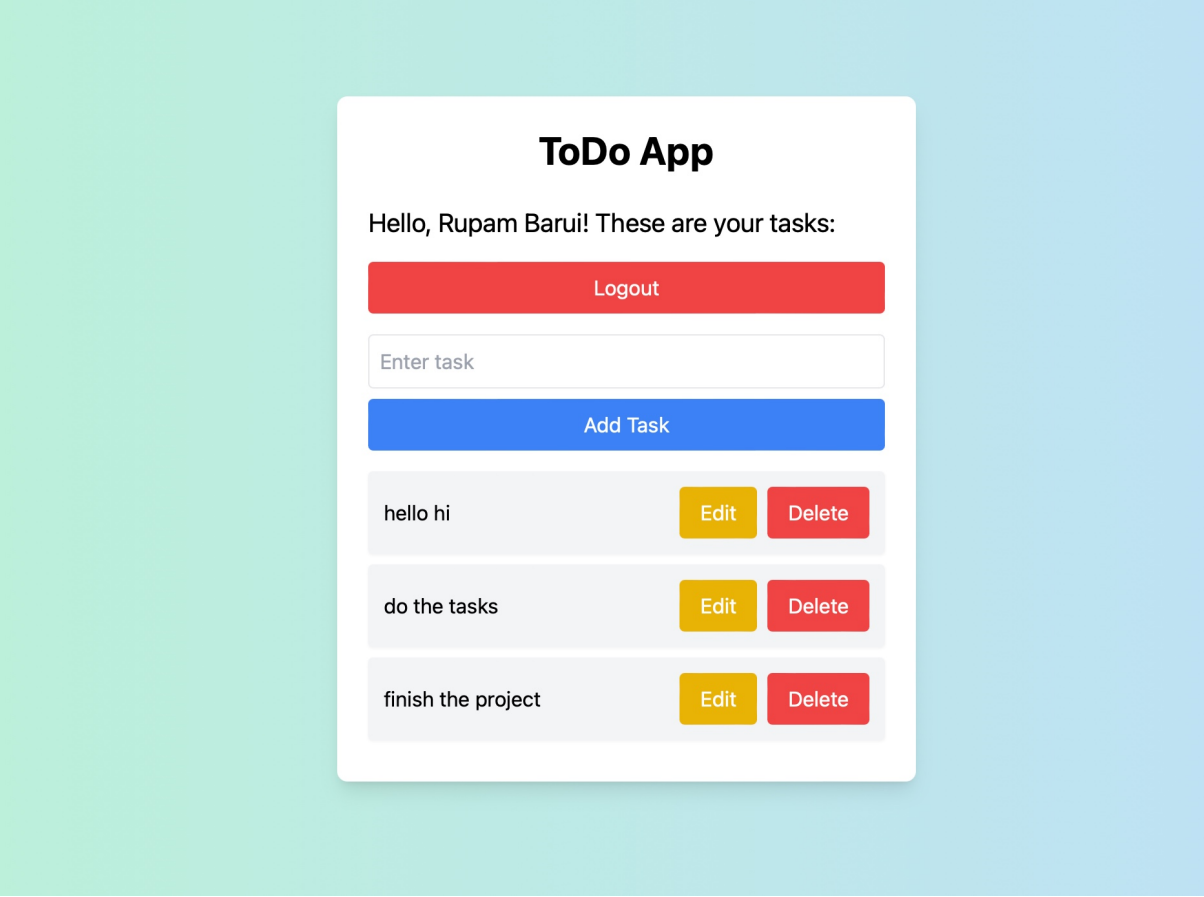

Figure 4: ToDo app with CRUD operations

Experiment No. 5: Set up a GraphQL Server

Objective

The objective of this experiment is to set up a GraphQL server using Node.js and Apollo Server to handle queries and mutations.

Introduction

GraphQL is a query language for APIs and a runtime for executing those queries by using a type system you define for your data. In this experiment, we will set up a GraphQL server using Node.js and Apollo Server to manage and serve data.

Theory

In this experiment, we will:

- Use Node.js to create a server.
- Use Apollo Server to handle GraphQL requests.
- Define a GraphQL schema with types, queries, and mutations.
- Implement resolver functions to handle the logic for each query and mutation.
- Test the GraphQL server using GraphQL Playground or a similar tool.

Algorithm

1. Set up a new Node.js project.
2. Install necessary packages like ‘apollo-server‘, ‘graphql‘, and ‘express‘.
3. Define a GraphQL schema with types, queries, and mutations.
4. Implement resolver functions for each query and mutation.
5. Set up Apollo Server to use the schema and resolvers.
6. Test the GraphQL server using GraphQL Playground.

Program

```
// app.js
const { ApolloServer, gql } = require('apollo-server');
const { v4: uuidv4 } = require('uuid');

// Sample data
```

```
let todos = [
  { id: uuidv4(), title: "Learn GraphQL", completed: false },
  { id: uuidv4(), title: "Build a GraphQL server", completed: false }
];

// Type definitions
const typeDefs = gql`
  type Todo {
    id: ID!
    title: String!
    completed: Boolean!
  }

  type Query {
    getTodos: [Todo]
    getTodoById(id: ID!): Todo
  }

  type Mutation {
    addTodo(title: String!): Todo
    updateTodo(id: ID!, title: String, completed: Boolean): Todo
    deleteTodo(id: ID!): String
  }
`;

// Resolvers
const resolvers = {
  Query: {
    getTodos: () => todos,
    getTodoById: (parent, args) => todos.find(todo => todo.id === args.id)
  },
  Mutation: {
    addTodo: (parent, args) => {
      const newTodo = { id: uuidv4(), title: args.title, completed: false };
      todos.push(newTodo);
      return newTodo;
    },
    updateTodo: (parent, args) => {
      const todo = todos.find(todo => todo.id === args.id);
      if (!todo) {
        throw new Error("Todo not found");
      }
      if (args.title !== undefined) {
        todo.title = args.title;
      }
      if (args.completed !== undefined) {
```

```
        todo.completed = args.completed;
      }
      return todo;
    },
    deleteTodo: (parent, args) => {
      const index = todos.findIndex(todo => todo.id === args.id);
      if (index === -1) {
        throw new Error("Todo not found");
      }
      todos.splice(index, 1);
      return "Todo deleted";
    }
  }
};

// Apollo Server setup
const server = new ApolloServer({ typeDefs, resolvers });

server.listen().then(({ url }) => {
  console.log(`Server is running at ${url}`);
});
```

Conclusion

In this experiment, we successfully set up a GraphQL server using Node.js and Apollo Server. We defined a GraphQL schema and implemented resolver functions to handle queries and mutations, demonstrating the basic principles of creating a GraphQL API.

Figure 5: Output of the GraphQL server showing queries

Figure 6: Output of the GraphQL server showing mutations

Figure 7: Output of the GraphQL server showing update

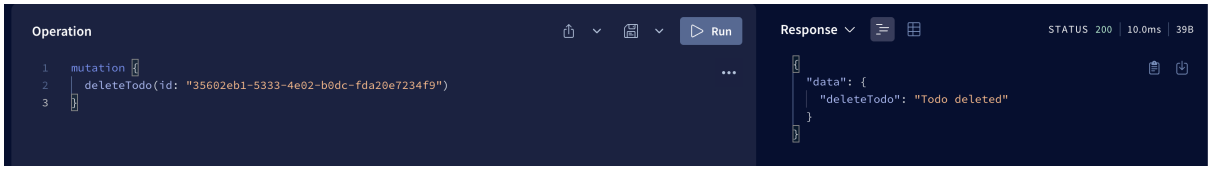

Figure 8: Output of the GraphQL server showing delete

Experiment No. 6: To create React Component

Objective

The objective of this experiment is to create a simple React component that displays a list of items.

Introduction

React is a JavaScript library for building user interfaces. Components are the building blocks of a React application. In this experiment, we will create a simple React component to display a list of items.

Theory

In this experiment, we will:

- Use Create React App to set up a new React project.
- Define a functional component in React.
- Use React props to pass data to the component.
- Render a list of items using the component.

Algorithm

1. Set up a new React project using Create React App.
2. Create a functional component that accepts props.
3. Use the props to render a list of items.
4. Integrate the component into the main application component.
5. Run the React application to view the component.

Program

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

// ItemList Component
const ItemList = ({ items, color }) => {
  return (
    <ul className="list-disc pl-5 space-y-2">
      {items.map((item, index) => (
        <li
          key={index}
```

```
                className={`${color} p-4 m-2 rounded-lg`}
            >
                {item}
            </li>
        ))}
    </ul>
  );
};

// App Component
const App = () => {
  const [numItems, setNumItems] = useState(4);
  const [color, setColor] = useState("bg-blue-200");

  const handleNumItemsChange = (e) => {
    setNumItems(Number(e.target.value));
  };

  const handleColorChange = (e) => {
    setColor(e.target.value);
  };

  const items = Array.from(
    { length: numItems },
    (_, index) => `Item ${index + 1}`
  );

  return (
    <div className="App min-h-screen bg-gray-100 flex flex-col items-center
        justify-center space-y-4">
      <div className="bg-white shadow-lg rounded-lg p-8">
        <h1 className="text-2xl font-bold mb-4">Item List</h1>
        <div className="mb-4">
          <label className="block mb-2 font-semibold">
            Number of Items:
          </label>
          <input
            type="number"
            min="1"
            value={numItems}
            onChange={handleNumItemsChange}
            className="border rounded p-2 w-full"
          />
        </div>
        <div className="mb-4">
          <label className="block mb-2 font-semibold">
```

```
                Item Color:
              </label>
              <select
                value={color}
                onChange={handleColorChange}
                className="border rounded p-2 w-full"
              >
                <option value="bg-red-200">Red</option>
                <option value="bg-blue-200">Blue</option>
                <option value="bg-green-200">Green</option>
                <option value="bg-yellow-200">Yellow</option>
              </select>
            </div>
            <ItemList
              items={items}
              color={color}
            />
          </div>
        </div>
    );
};

export default App;
```

Conclusion

In this experiment, we successfully created a simple React component to display a list of items. We used React props to pass data to the component and rendered the list within the component.

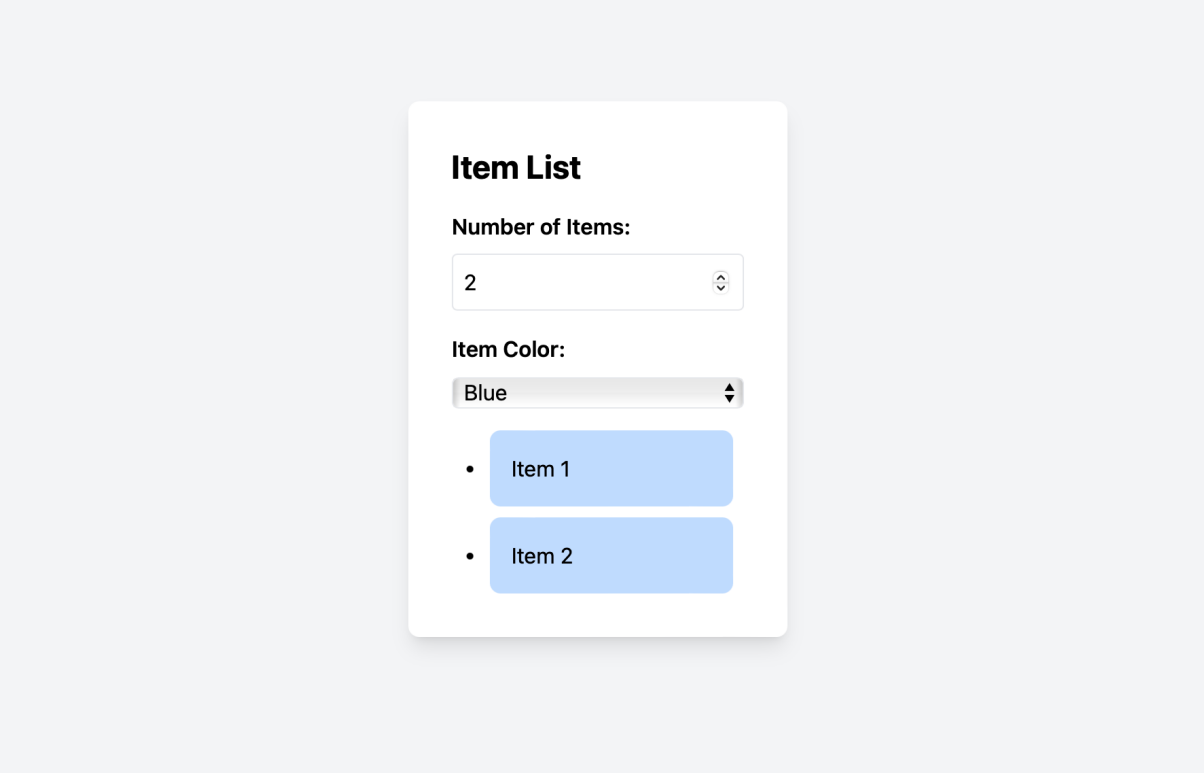

Figure 9: React Component

Experiment No. 7: Implement State and Props in React

Objective

The objective of this experiment is to demonstrate the use of state and props in a React component.

Introduction

React is a JavaScript library for building user interfaces. State and props are two core concepts in React that help manage data in components. State allows components to manage and update their own data, while props allow data to be passed from parent components to child components.

Theory

In this experiment, we will:

- Use React state to manage the data within a component.
- Use React props to pass data from a parent component to a child component.
- Update the state based on user interactions.

Algorithm

1. Set up a new React project using Create React App.
2. Create a functional component with state using the ‘useState‘ hook.
3. Pass the state and state-updating function as props to a child component.
4. Update the state based on user input in the child component.
5. Render the updated state in the parent component.

Program

```
// App.js
import React, { useState } from "react";

// Child Component
const ChildComponent = ({ count, incrementCount }) => {
   return (
      <div className="bg-gray-100 rounded-lg p-4 shadow-md">
         <p className="text-xl font-bold">Count: {count}</p>
         <button
```

```
            className="bg-blue-500 hover:bg-blue-600 text-white font-bold py-2
                px-4 rounded mt-2"
            onClick={incrementCount}
        >
            Increment
        </button>
    </div>
  );
};

// App Component
const App = () => {
  const [count, setCount] = useState(0);

  const incrementCount = () => {
    setCount(count + 1);
  };

  return (
    <div className="bg-gray-200 min-h-screen flex items-center
        justify-center">
      <div className="max-w-lg w-full p-8 bg-white rounded-lg shadow-md">
        <h1 className="text-3xl font-bold mb-4">
          State and Props in React
        </h1>
        <ChildComponent
          count={count}
          incrementCount={incrementCount}
        />
      </div>
    </div>
  );
};

export default App;
```

Conclusion

In this experiment, we successfully demonstrated the use of state and props in a React component. We used the ‘useState‘ hook to manage state in a parent component and passed the state and a state-updating function as props to a child component. The child component updated the state based on user interaction, and the updated state was rendered in the parent component.

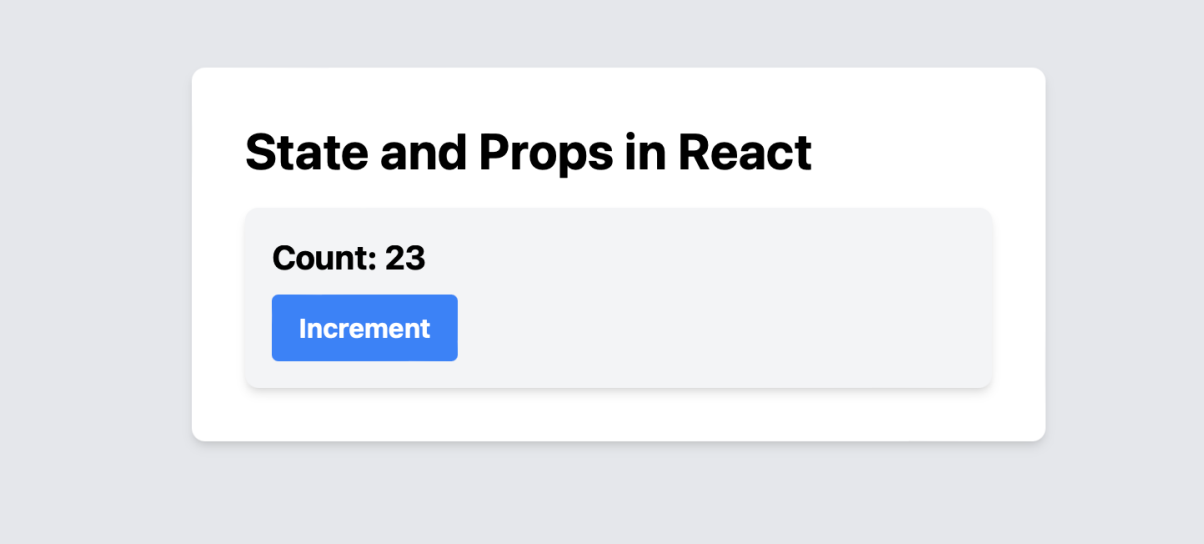

Figure 10: States and props in react

Experiment No. 8: Implement Hooks (useState and useEffect) in React

Objective

The objective of this experiment is to demonstrate the use of the ‘useState‘ and ‘useEffect‘ hooks in a React component.

Introduction

React Hooks are functions that let us use state and other React features in functional components. The ‘useState‘ hook allows us to add state to functional components, and the ‘useEffect‘ hook lets us perform side effects in functional components.

Theory

In this experiment, we will:

- Use the ‘useState‘ hook to manage state within a functional component.
- Use the ‘useEffect‘ hook to perform side effects, such as fetching data or directly manipulating the DOM, after the component renders.
- Understand the dependency array in the ‘useEffect‘ hook to control when the side effect should run.

Algorithm

1. Set up a new React project using Create React App.
2. Create a functional component that uses the ‘useState‘ hook to manage a counter.
3. Use the ‘useEffect‘ hook to update a message displayed in the component every time the counter updates.
4. Render the counter value, the message, and a button to increment the counter in the component.

Program

```
// App.js
import React, { useEffect, useState } from "react";

const App = () => {
   const [count, setCount] = useState(0);
   const [message, setMessage] = useState("");

   useEffect(() => {
```

```
      setMessage(`Count has been updated to: ${count}`);
   }, [count]); // The effect will run only when 'count' changes

   return (
      <div className="App min-h-screen flex flex-col items-center
          justify-center text-black">
         <h1 className="text-4xl font-bold mb-4">
            Using useState and useEffect Hooks in React
         </h1>
         <p className="text-2xl mb-4">Count: {count}</p>
         <button
            className="bg-blue-700 hover:bg-blue-900 text-white font-bold py-2
               px-4 rounded-full mb-4 transition duration-300 ease-in-out"
            onClick={() => setCount(count + 1)}
         >
            Increment
         </button>
         <p className="text-lg">{message}</p>
      </div>
   );
};

export default App;
```

Conclusion

In this experiment, we successfully demonstrated the use of the `useState` and `useEffect` hooks in a React component. We used the `useState` hook to manage the state of a counter and the `useEffect` hook to update a message displayed in the component whenever the counter value was updated. This experiment highlights how hooks can be used to manage state and side effects in functional components.

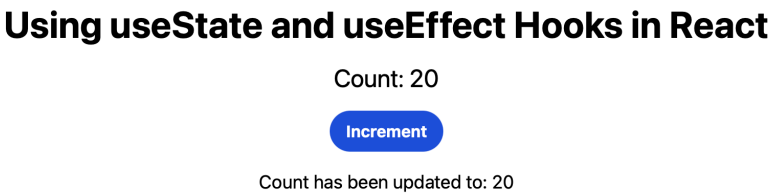

Figure 11: Output of the React component demonstrating the use of useState and useEffect hooks

Experiment No. 9: How to handle form elements in React

Objective

The objective of this experiment is to demonstrate how to handle form elements in a React component, including managing form state and handling form submission.

Introduction

Handling form elements in React involves managing the state of input fields and updating the state based on user interactions. React provides controlled components where form data is handled by the state within the component. This allows for better control over form inputs and validation.

Theory

In this experiment, we will:

- Use controlled components to manage the state of form inputs.
- Handle form submission and prevent the default form submission behavior.
- Update state based on user inputs and display the entered data.
- Understand the importance of managing form state in React for maintaining the flow of data in the application.

Algorithm

1. Set up a new React project using Create React App.
2. Create a functional component with form elements (e.g., input fields, textarea, select).
3. Use the ‘useState‘ hook to manage the state of each form element.
4. Handle form submission by creating a function that processes the form data and prevents the default submission behavior.
5. Render the form elements and display the entered data below the form.

Program

```
import React, { useState } from 'react';
import 'tailwindcss/tailwind.css'; // Ensure Tailwind CSS is included in your
    project

const App = () => {
  const [formData, setFormData] = useState({
```

```
    name: '',
    email: '',
    message: ''
  });

  const handleChange = (e) => {
    const { name, value } = e.target;
    setFormData({
      ...formData,
      [name]: value
    });
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    alert(`Form submitted: ${JSON.stringify(formData)}`);
  };

  return (
    <div className="min-h-screen flex flex-col items-center justify-center
        bg-gray-100">
      <h1 className="text-3xl font-bold mb-8 text-gray-700">Handling Form
          Elements in React</h1>
      <form onSubmit={handleSubmit} className="bg-white shadow-md rounded px-8
          pt-6 pb-8 mb-4">
        <div className="mb-4">
          <label className="block text-gray-700 text-sm font-bold mb-2">
            Name:
            <input
              type="text"
              name="name"
              value={formData.name}
              onChange={handleChange}
              className="shadow appearance-none border rounded w-full py-2 px-3
                  text-gray-700 leading-tight focus:outline-none
                  focus:shadow-outline"
            />
          </label>
        </div>
        <div className="mb-4">
          <label className="block text-gray-700 text-sm font-bold mb-2">
            Email:
            <input
              type="email"
              name="email"
              value={formData.email}
```

```
            onChange={handleChange}
            className="shadow appearance-none border rounded w-full py-2 px-3
                text-gray-700 leading-tight focus:outline-none
                focus:shadow-outline"
          />
        </label>
      </div>
      <div className="mb-6">
        <label className="block text-gray-700 text-sm font-bold mb-2">
          Message:
          <textarea
            name="message"
            value={formData.message}
            onChange={handleChange}
            className="shadow appearance-none border rounded w-full py-2 px-3
                text-gray-700 leading-tight focus:outline-none
                focus:shadow-outline"
          />
        </label>
      </div>
      <div className="flex items-center justify-between">
        <button
          type="submit"
          className="bg-blue-500 hover:bg-blue-700 text-white font-bold py-2
              px-4 rounded focus:outline-none focus:shadow-outline"
        >
          Submit
        </button>
      </div>
    </form>
    <div className="bg-white shadow-md rounded px-8 pt-6 pb-8 mb-4 mt-8
        w-full max-w-lg">
      <h2 className="text-2xl font-bold mb-4 text-gray-700">Form Data:</h2>
      <p className="text-gray-700"><strong>Name:</strong> {formData.name}</p>
      <p className="text-gray-700"><strong>Email:</strong>
          {formData.email}</p>
      <p className="text-gray-700"><strong>Message:</strong>
          {formData.message}</p>
    </div>
  </div>
 );
};

export default App;
```

Conclusion

In this experiment, we successfully demonstrated how to handle form elements in a React component. We used controlled components to manage the state of form inputs and handled form submission to process and display the entered data. This experiment highlights the importance of managing form state in React for creating dynamic and interactive forms.

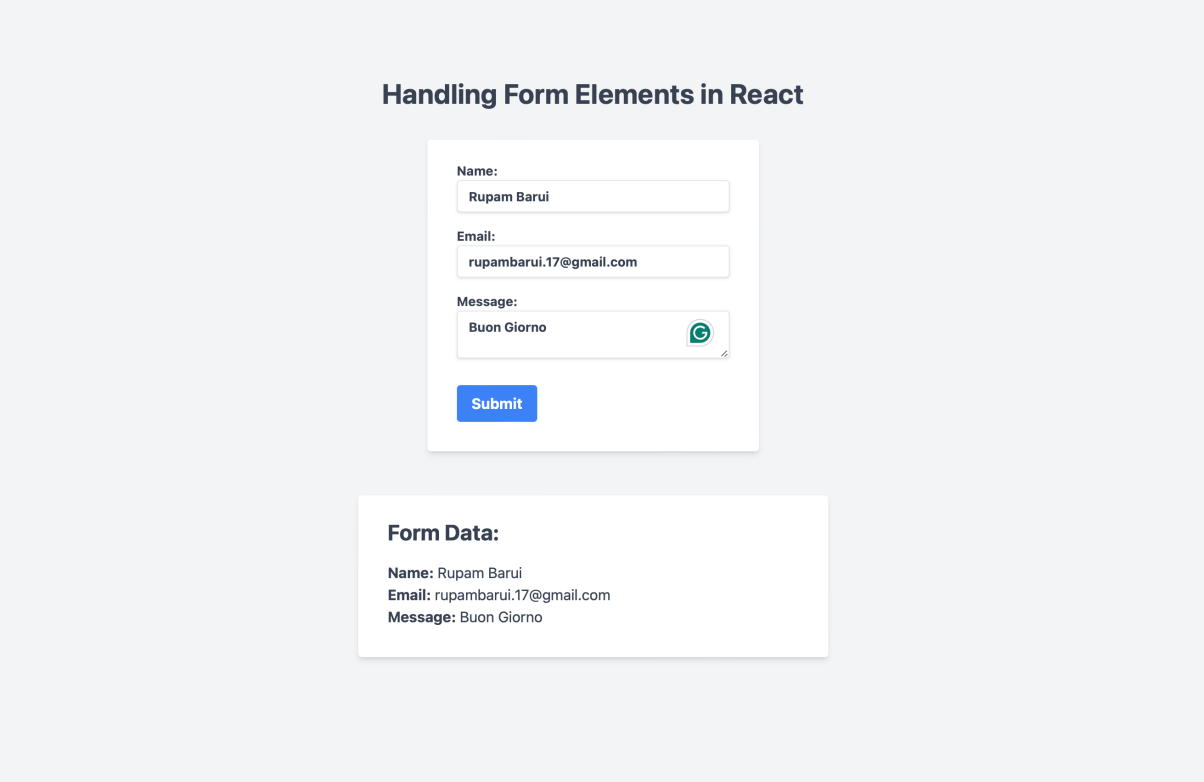

Figure 12: React component demonstrating form handling

Experiment No. 10: Implement Node.js events

Objective

The objective of this experiment is to demonstrate how to implement and handle events in a Node.js application using the EventEmitter class.

Introduction

Node.js is built on an event-driven architecture, making it well-suited for applications that require asynchronous operations. The EventEmitter class in Node.js provides a way to handle custom events, allowing developers to define, emit, and handle events in their applications.

Theory

In this experiment, we will:

- Use the EventEmitter class to create an event emitter.
- Define custom events and their corresponding event handlers.
- Emit events and observe how the event handlers respond to them.
- Understand the role of event-driven programming in Node.js applications.

Algorithm

1. Set up a new Node.js project.
2. Import the EventEmitter class from the 'events' module.
3. Create an instance of EventEmitter.
4. Define custom events and their handlers using the 'on' method.
5. Emit events using the 'emit' method and observe the output.

Program

```
const EventEmitter = require('events');

class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();

// Event listener
myEmitter.on('event', () => {
  console.log('An event occurred!');
```

```
});

// Emit the event
myEmitter.emit('event');

// Event with arguments
myEmitter.on('eventWithArgs', (arg1, arg2) => {
  console.log(`Event with args: ${arg1}, ${arg2}`);
});

// Emit event with arguments
myEmitter.emit('eventWithArgs', 'Hello', 'World');
```

Conclusion

In this experiment, we successfully demonstrated how to implement and handle custom events in a Node.js application using the EventEmitter class. We defined custom events, created event handlers, and emitted events to observe the behavior of our application. This experiment highlights the importance of event-driven programming in building efficient and responsive Node.js applications.

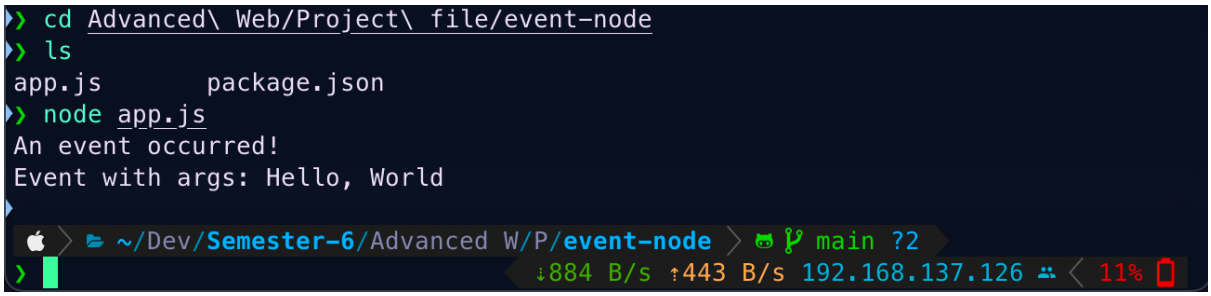

Figure 13: Output of the Node.js events implementation

Experiment No. 11: Pass UserID to Frontend with Firebase

Objective

The objective of this experiment is to demonstrate how to pass the ‘userID‘ to the frontend using Firebase authentication and Firestore, allowing for user-specific data handling and displaying mood journal entries on a graph.

Introduction

In this experiment, we will use Firebase for authentication and Firestore as the database to store user-specific mood journal entries. The user’s ID will be used to fetch and manipulate data related to the logged-in user, and this information will be displayed on the frontend along with a graph to visualize the user’s mood over time.

Theory

In this experiment, we will:

- Set up Firebase authentication to allow users to log in.
- Use Firestore to store mood journal entries with a reference to the ‘userID‘.
- Pass the ‘userID‘ to the frontend to fetch and display mood journal entries related to the logged-in user.
- Display the mood journal entries on a graph.

Algorithm

1. Set up a new Firebase project and integrate Firebase into the React app.
2. Implement Google authentication using Firebase.
3. Use Firestore to store mood journal entries with the date, time, and mood.
4. Fetch and display mood journal entries specific to the logged-in user.
5. Display a graph visualizing the user’s mood over time.

Program

App.js:

```
import { initializeApp } from "firebase/app";
import {
    GoogleAuthProvider,
    getAuth,
    signInWithPopup,
```

```
    signOut,
} from "firebase/auth";
import {
    addDoc,
    collection,
    deleteDoc,
    doc,
    getFirestore,
    onSnapshot,
    query,
    where,
} from "firebase/firestore";
import React, { useEffect, useState } from "react";
import { useAuthState } from "react-firebase-hooks/auth";
import { Line } from 'react-chartjs-2';
import 'chart.js/auto';

const firebaseConfig = {
    apiKey: "YOUR_API_KEY",
    authDomain: "YOUR_AUTH_DOMAIN",
    projectId: "YOUR_PROJECT_ID",
    storageBucket: "YOUR_STORAGE_BUCKET",
    messagingSenderId: "YOUR_MESSAGING_SENDER_ID",
    appId: "YOUR_APP_ID",
};

initializeApp(firebaseConfig);
const auth = getAuth();
const provider = new GoogleAuthProvider();
const db = getFirestore();

const App = () => {
    const [user] = useAuthState(auth);
    const [entries, setEntries] = useState([]);
    const [date, setDate] = useState("");
    const [time, setTime] = useState("");
    const [mood, setMood] = useState("");

    const moodOptions = ["Happy", "Sad", "Angry", "Excited", "Neutral"];

    useEffect(() => {
        if (user) {
            const q = query(
                collection(db, "moodEntries"),
                where("userId", "==", user.uid)
            );
```

```
        const unsubscribe = onSnapshot(q, (querySnapshot) => {
            const entriesData = [];
            querySnapshot.forEach((doc) => {
                entriesData.push({ ...doc.data(), id: doc.id });
            });
            setEntries(entriesData);
        });
        return () => unsubscribe();
    }
}, [user]);

const handleLogin = async () => {
    await signInWithPopup(auth, provider);
};

const handleLogout = () => {
    signOut(auth);
};

const addEntry = async () => {
    if (!date || !time || !mood) return;
    await addDoc(collection(db, "moodEntries"), {
        date,
        time,
        mood,
        createdAt: new Date(),
        userId: user.uid,
    });
    setDate("");
    setTime("");
    setMood("");
};

const deleteEntry = async (id) => {
    await deleteDoc(doc(db, "moodEntries", id));
};

const moodData = {
    labels: entries.map(entry => `${entry.date} ${entry.time}`),
    datasets: [{
        label: 'Mood',
        data: entries.map(entry => moodOptions.indexOf(entry.mood) + 1),
        fill: false,
        borderColor: 'blue',
    }]
};
```

```
return (
    <div className="min-h-screen bg-gradient-to-r from-green-200
       to-blue-200 flex flex-col items-center justify-center p-4">
       <div className="bg-white p-6 rounded-lg shadow-lg w-full max-w-md">
          <h1 className="text-3xl font-bold mb-6 text-center">
             Mood Journal
          </h1>
          {user ? (
             <>
                <p className="text-xl mb-4">
                   Hello, {user.displayName}!
                </p>
                <p className="text-sm mb-4">UserID: {user.uid}</p>
                <button
                   onClick={handleLogout}
                   className="bg-red-500 text-white px-4 py-2 rounded
                      hover:bg-red-600 mb-4 w-full"
                >
                   Logout
                </button>
                <div className="mb-4">
                   <input
                      type="date"
                      className="border p-2 w-full rounded mb-2"
                      value={date}
                      onChange={(e) => setDate(e.target.value)}
                   />
                   <input
                      type="time"
                      className="border p-2 w-full rounded mb-2"
                      value={time}
                      onChange={(e) => setTime(e.target.value)}
                   />
                   <select
                      className="border p-2 w-full rounded mb-2"
                      value={mood}
                      onChange={(e) => setMood(e.target.value)}
                   >
                      <option value="">Select Mood</option>
                      {moodOptions.map(option => (
                         <option key={option} value={option}>
                            {option}
                         </option>
                      ))}
                   </select>
```

```
                        <button
                            onClick={addEntry}
                            className="bg-blue-500 text-white px-4 py-2
                                rounded w-full"
                        >
                            Add Entry
                        </button>
                    </div>
                    <ul>
                        {entries.map((entry) => (
                            <li
                                key={entry.id}
                                className="flex justify-between items-center
                                    bg-gray-100 p-3 my-2 rounded shadow-sm"
                            >
                                <span>{`${entry.date} ${entry.time} -
                                    ${entry.mood}`}</span>
                                <button
                                    onClick={() => deleteEntry(entry.id)}
                                    className="bg-red-500 text-white px-4
                                        py-2 rounded hover:bg-red-600"
                                >
                                    Delete
                                </button>
                            </li>
                        ))}
                    </ul>
                    <div className="mt-6">
                        <Line data={moodData} />
                    </div>
                </>
            ) : (
                <button
                    onClick={handleLogin}
                    className="bg-green-500 text-white px-4 py-2 rounded
                        hover:bg-green-600 w-full"
                >
                    Login with Google
                </button>
            )}
        </div>
    </div>
  );
};

export default App;
```

Conclusion

In this experiment, we successfully implemented a Mood Journal application using Firebase authentication and Firestore. The application allows users to log in, add mood entries with date and time, and visualize their mood over time on a graph. This demonstrates how user-specific data can be managed and displayed effectively in a React application with Firebase.

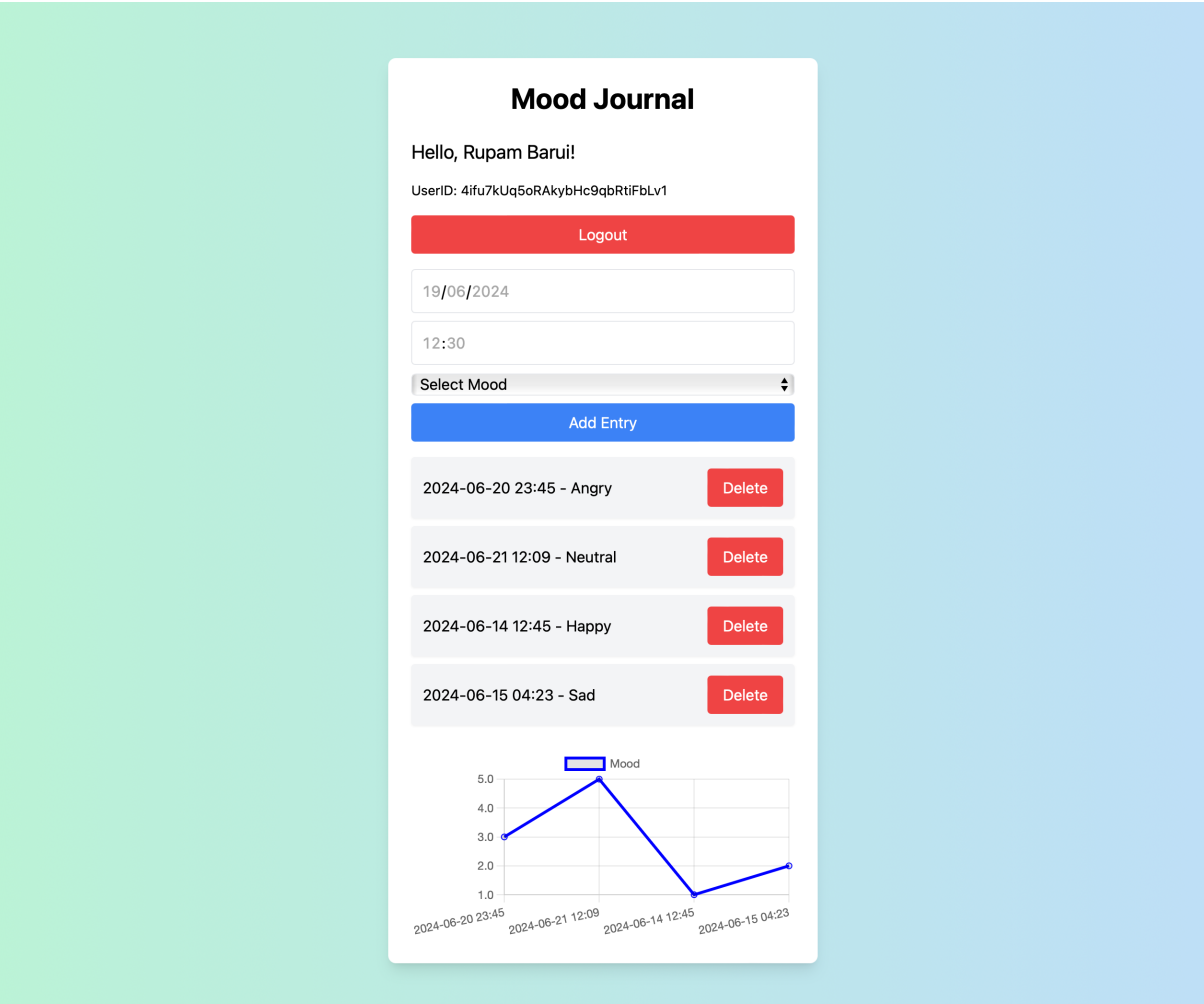

Figure 14: User Id Passing in Front-end