REACT

1) . State Management (Redux, Redux-Toolkit or Recoil)

THEORY EXERCISE

Question 1: What is Redux, and why is it used in React applications? Explain the core concepts of actions, reducers, and the store.

Redux is a **state management library** commonly used in React applications. It provides a centralized place (called the store) to manage the state of an application, making it easier to track and update state consistently across components. Redux is particularly useful in large or complex applications where managing state across multiple components can become challenging.

Why Use Redux in React Applications?

- **Centralized State Management:** Redux stores the entire application's state in a single store, which makes debugging and state management easier.
- **Predictable State Changes:** State changes in Redux follow a strict unidirectional flow, ensuring consistency and predictability.
- **Ease of Debugging:** With tools like Redux DevTools, you can track every state change, inspect actions, and even time-travel through the app's state.
- **Component Decoupling:** By using Redux, components don't need to pass state or props through multiple layers, reducing dependency between components.

Core Concepts of Redux

1) Store

- The **store** is the central repository for all state in a Redux application.
- It holds the entire state tree and provides methods for accessing the state, dispatching actions, and subscribing to state changes.
- A Redux application typically has only one store.
- import { createStore } from "redux";

  const store = createStore(reducer);

2) Actions

- Actions are plain JavaScript objects that describe what should happen in the application.
- Every action has a type property that describes the type of event, and it may also include additional payload data.

- Actions are dispatched to the store to initiate state changes.
-const incrementAction = { type: "INCREMENT" }; const addUserAction = { type: "ADD_USER", payload: { name: "John Doe" } };

3) **Reducers**
- Reducers are pure functions that specify how the application's state should change in response to an action.
- A reducer takes the current state and an action as arguments and returns the new state.
- const initialState = { count: 0 };

```
const counterReducer = (state = initialState, action) => {
  switch (action.type) {
    case "INCREMENT":
      return { ...state, count: state.count + 1 };
    case "DECREMENT":
      return { ...state, count: state.count - 1 };
    default:
      return state;
  }
};
```

Redux Flow

Dispatch an Action:
A user interaction (e.g., button click) triggers an action dispatch.

Reducers Handle the Action:
The store passes the current state and the dispatched action to the reducer.

Update the Store:
The reducer returns a new state, which updates the store.

React Components React to Changes:
Components subscribed to the store re-render with the updated state.

-EXAMPLE

2) Set up redux store

```
// store.js
import { createStore } from "redux";

const initialState = { count: 0 };

const counterReducer = (state = initialState, action) => {
  switch (action.type) {
    case "INCREMENT":
      return { count: state.count + 1 };
    case "DECREMENT":
      return { count: state.count - 1 };
    default:
      return state;
  }
};
const store = createStore(counterReducer);

export default store;
```

3) Connect React with Redux:

```
// index.js
import React from "react";
import ReactDOM from "react-dom";
import { Provider } from "react-redux";
import App from "./App";
import store from "./store";

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById("root")
);
```

4) Access Redux State in a Component:

```
// App.js
import React from "react";
import { useSelector, useDispatch } from
"react-redux";

const App = () => {
  const count = useSelector((state) =>
state.count);
  const dispatch = useDispatch();

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={() => dispatch({
type: "INCREMENT"
})}>Increment</button>
      <button onClick={() => dispatch({
type: "DECREMENT"
})}>Decrement</button>
    </div>
  );
};

export default App;
```

Question 2: How does Recoil simplify state management in React compared to Redux?

Recoil is a modern state management library designed specifically for React. It simplifies state management by tightly integrating with React's component model, offering a more intuitive and minimalistic approach compared to **Redux**. Here's how Recoil simplifies state management:

1. Direct Integration with React

Recoil:

- Works seamlessly with React's state and hooks, making it feel like an extension of React itself.
- Components can directly access and modify state using useRecoilState, similar to React's useState.
- No need for external boilerplate like actions, reducers, or a store.
- EX

```
import { atom, useRecoilState } from "recoil";

const countState = atom({
  key: "countState",
  default: 0,
});

const Counter = () => {
  const [count, setCount] = useRecoilState(countState);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
};
```

Redux:

- Requires setting up a global store, reducers, and actions.
- State management logic is separated from components, requiring more effort to connect components to the store using useSelector and useDispatch.

2. Reduced Boilerplate

Recoil:

- Focuses on simplicity with no need for creating actions or reducers.
- State is represented as atoms (pieces of state) or selectors (derived state).
- EX :

```
const countState = atom({ key: "countState", default: 0 });
```

Redux:

- Involves more boilerplate, including creating a store, reducers, action types, and dispatching actions.
- EX

```
const INCREMENT = "INCREMENT";

const incrementAction = () => ({ type: INCREMENT });

const counterReducer = (state = 0, action) => {
  switch (action.type) {
    case INCREMENT:
      return state + 1;
    default:
      return state;
  }
};
```

3. Component-Specific State Management

Recoil:

- State is stored in individual atoms, allowing components to subscribe only to the atoms they care about.
- When an atom's state changes, only components subscribed to that atom re-render.
- EX:

```
const countState = atom({ key: "countState", default: 0 });
const textState = atom({ key: "textState", default: "" });
```

Redux:

- A centralized global store manages all state.
- Components often need to select and manage slices of the global state, which can make the state management logic more complex.

4. Built-In Derived State (Selectors)

Recoil:

- Provides selectors to derive state from atoms. Selectors are like computed properties that depend on one or more atoms.
- EX

```
import { selector } from "recoil";

const doubleCountState = selector({
  key: "doubleCountState",
  get: ({ get }) => get(countState) * 2,
});
```

Redux:

- Requires libraries like Reselect for deriving state.
- Adding selectors increases the amount of boilerplate and complexity.

5. Learning Curve

Recoil:

- Easier to learn and use, especially for React developers.
- Relies on concepts familiar to React developers, like hooks and component-based state.

Redux:

- Steeper learning curve due to the need to understand actions, reducers, middleware, and other concepts.

6. Local vs. Global State

Recoil:

- Designed to manage both **local** and **global** state seamlessly within the same system.
- Eliminates the need to distinguish between component state (useState) and global state (Redux).

Redux:

- Focuses primarily on managing global state, requiring additional libraries (like redux-localstorage) to manage local state effectively.

7. Reactivity and Performance

Recoil:

- Reactivity is built-in. Only components subscribed to an atom re-render when that atom changes.
- Optimized for performance out-of-the-box with fine-grained subscriptions.

Redux:

- Updates the entire state tree even if only a small part of the state changes, which can lead to unnecessary re-renders unless optimizations like memoization are used.

When to Use Recoil vs. Redux

	Recoil	Readux
Project Size	Small to medium applications	Medium to large applications
Boilerplate	Minimal	High
Learning Curve	Easy	Moderate to steep
State Dependencies	Complex state dependencies and derived state	Simple or predictable state structures
Performance Needs	High reactivity with fine-grained updates	Requires additional optimization

LAB EXERCISE

Task 1: o Create a simple counter application using Redux for state management. Implement actions to increment and decrement the counter.

- **Steps:**

1) **Set up Redux in React.**
2) **Create actions for increment and decrement.**
3) **Create a reducer to handle the counter logic**
4) **-Connect Redux to React using the Provider and useSelector/useDispatch hooks.**

// store.js

```
import { createStore } from "redux";
const counterReducer = (state = { count: 0 }, action) => {
  switch (action.type) {
    case "INCREMENT":
      return { count: state.count + 1 };
    case "DECREMENT":
      return { count: state.count - 1 };
    default:
      return state;
  }
};
// Create Store
const store = createStore(counterReducer);
export default store;
```

// App.js

```
import React from "react";
import { useSelector, useDispatch } from "react-redux";
```

```
const App = () => {
  const count = useSelector((state) => state.count);
  const dispatch = useDispatch();

  return (
    <div>
      <h1>Counter: {count}</h1>
      <button onClick={() => dispatch({ type: "INCREMENT" })}>Increment</button>
      <button onClick={() => dispatch({ type: "DECREMENT" })}>Decrement</button>
    </div>
  );
};
export default App;
```

```
  // index.js
import React from "react";
import ReactDOM from "react-dom";
import { Provider } from "react-redux";
import App from "./App";
import store from "./store";

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById("root")
);
```

- Connect Redux to React using the Provider and useSelector/useDispatch hooks.

Task 2: o Build a todo list application using Recoil for state management. Allow users to add, remove, and mark tasks as complete.

Steps:
Set up Recoil in the React app.
Create atoms for managing the todo list state.
Implement functionality to add, remove, and mark tasks as complete.

// index.js

```
import React from "react";
import ReactDOM from "react-dom";
import { RecoilRoot } from "recoil";
import App from "./App";

ReactDOM.render(
  <RecoilRoot>
    <App />
  </RecoilRoot>,
  document.getElementById("root")
);
```

// atoms.js

```
import { atom } from "recoil";

export const todoListState = atom({
  key: "todoListState",
  default: [],
});
```

// App.js

```
import React, { useState } from "react";
import { useRecoilState } from "recoil";
import { todoListState } from "./atoms";

const App = () => {
  const [todos, setTodos] = useRecoilState(todoListState);
  const [task, setTask] = useState("");

  const addTodo = () => {
    if (task) {
      setTodos([...todos, { id: Date.now(), text: task, completed: false }]);
      setTask("");
    }
  };

  const toggleComplete = (id) => {
    setTodos(
      todos.map((todo) =>
        todo.id === id ? { ...todo, completed: !todo.completed } : todo
      )
```

```
    );
  };

  const deleteTodo = (id) => {
    setTodos(todos.filter((todo) => todo.id !== id));
  };

  return (
    <div>
      <h1>Todo List</h1>
      <input
        type="text"
        value={task}
        onChange={(e) => setTask(e.target.value)}
        placeholder="Add a task"
      />
      <button onClick={addTodo}>Add</button>
      <ul>
        {todos.map((todo) => (
          <li key={todo.id}>
            <span
              style={{
                textDecoration: todo.completed ? "line-through" : "none",
              }}
              onClick={() => toggleComplete(todo.id)}
            >
              {todo.text}
            </span>
            <button onClick={() => deleteTodo(todo.id)}>Delete</button>
          </li>
        ))}
      </ul>
    </div>
  );
};

export default App;
```

Task 3: o Build a crud application using Redux-Toolkit for state management. Allow users to add,remove, delete and update

Steps:
Set up Redux-Toolkit in a React app.
Create a slice for managing CRUD operations.
Connect the slice to the store.
Implement actions for adding, removing, deleting, and updating items.

// store.js

```
import { configureStore } from "@reduxjs/toolkit";
import todoReducer from "./todoSlice";

export const store = configureStore({
  reducer: {
    todos: todoReducer,
  },
});
```

// todoSlice.js

```
import { createSlice } from "@reduxjs/toolkit";
const todoSlice = createSlice({
  name: "todos",
  initialState: [],
  reducers: {
    addTodo: (state, action) => {
      state.push({ id: Date.now(), text: action.payload, completed: false });
    },
    deleteTodo: (state, action) => {
      return state.filter((todo) => todo.id !== action.payload);
    },
    updateTodo: (state, action) => {
      const { id, newText } = action.payload;
      const todo = state.find((todo) => todo.id === id);
      if (todo) {
        todo.text = newText;
      }
    },
    toggleComplete: (state, action) => {
      const todo = state.find((todo) => todo.id === action.payload);
      if (todo) {
        todo.completed = !todo.completed;
      }
    },
  },
});
```

```
export const { addTodo, deleteTodo, updateTodo, toggleComplete } =
  todoSlice.actions;
export default todoSlice.reducer;
```

// App.js

```
import React, { useState } from "react";
import { useSelector, useDispatch } from "react-redux";
import { addTodo, deleteTodo, updateTodo, toggleComplete } from "./todoSlice";

const App = () => {
  const todos = useSelector((state) => state.todos);
  const dispatch = useDispatch();
  const [task, setTask] = useState("");
  const [editId, setEditId] = useState(null);
  const [newText, setNewText] = useState("");

  const handleAdd = () => {
    if (task) {
      dispatch(addTodo(task));
      setTask("");
    }
  };

  const handleUpdate = () => {
    if (newText) {
      dispatch(updateTodo({ id: editId, newText }));
      setEditId(null);
      setNewText("");
    }
  };

  return (
    <div>
      <h1>CRUD App</h1>
      <input
        type="text"
        value={task}
        onChange={(e) => setTask(e.target.value)}
        placeholder="Add a task"
      />
      <button onClick={handleAdd}>Add</button>
      <ul>
        {todos.map((todo) => (
          <li key={todo.id}>
            {editId === todo.id ? (
              <>
                <input
                  type="text"
                  value={newText}
```

```
            onChange={(e) => setNewText(e.target.value)}
           />
           <button onClick={handleUpdate}>Save</button>
          </>
         ) : (
          <>
           <span
            style={{
             textDecoration: todo.completed ? "line-through" : "none",
            }}
            onClick={() => dispatch(toggleComplete(todo.id))}
           >
            {todo.text}
           </span>
           <button onClick={() => setEditId(todo.id)}>Edit</button>
           <button onClick={() => dispatch(deleteTodo(todo.id))}>
            Delete
           </button>
          </>
         )}
        </li>
      ))}
    </ul>
   </div>
  );
};

export default App;
```

React – JSON-server and Firebase Real Time Database

THEORY EXERCISE

- ## Question 1:What do you mean by RESTful web services?

- RESTful web services are web services that conform to the REST (Representational State Transfer) architectural style. REST is a set of principles for designing networked applications, emphasizing a stateless, client-server model that leverages HTTP methods.
- Key features of RESTful web services:
- **Statelessness:** Each request from a client contains all the information the server needs to process it, without relying on stored context.
- **Resource-Based:** RESTful services expose resources, identified by URIs (e.g., /users or /products).
- **Standard HTTP Methods:** RESTful APIs use HTTP methods like:
  - GET (retrieve data)
  - POST (create data)
  - PUT or PATCH (update data)
  - DELETE (delete data)
- **Data Format:** Typically use JSON or XML for data exchange.
- **Scalability and Modularity:** RESTful APIs are easy to scale and maintain due to their stateless and modular nature.

Question 2: What is JSON-Server? How do we use it in React?

JSON-Server is a lightweight, full fake REST API generator. It allows developers to quickly create a mock API by serving a JSON file as a RESTful API endpoint. It's widely used for prototyping, testing, and front-end development.

Steps to use JSON-Server in React:

1) Install JSON-Server:

- **npm install -g json-server**

2) Create a db.json file:

- This file serves as the mock database]
- **EX**

```
{
 "users": [
  { "id": 1, "name": "John Doe" },
  { "id": 2, "name": "Jane Doe" }
 ]
}
```

3) Start the server:

- **json-server --watch db.json --port 5000**

4) Fetch data in React:

- Use fetch() or axios to make API requests to http://localhost:5000/users.

Question 3: How do you fetch data from a JSON-server API in React? Explain the role of fetch() or axios() in making API requests.

To fetch data from a JSON-server API in React, you can use the fetch() method or the axios library.

Example using fetch():

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

```
const App = () => {
  const [users, setUsers] = useState([]);
  useEffect(() => {
    fetch("http://localhost:5000/users")
      .then(response => response.json())
      .then(data => setUsers(data))
      .catch(error => console.error("Error:", error));
  }, []);
  return (
    <div>
      <h1>Users</h1>
      <ul>
        {users.map(user => (
          <li key={user.id}>{user.name}</li>
        ))}
      </ul>
    </div>
  );
};
export default App;
```

Example using axios:

- npm install axios

```
import React, { useEffect, useState } from "react";
import axios from "axios";
const App = () => {
  const [users, setUsers] = useState([]);
  useEffect(() => {
    axios
      .get("http://localhost:5000/users")
      .then(response => setUsers(response.data))
      .catch(error => console.error("Error:", error));
  }, []);
  return (
    <div>
```

```
 <h1>Users</h1>
     <ul>
       {users.map(user => (
         <li key={user.id}>{user.name}</li>
       ))}
     </ul>
   </div>
  );
};

export default App;
```

Role of fetch() or axios() in making API requests:

fetch():
A built-in JavaScript function for making HTTP requests. It returns a Promise and requires manual error handling.

axios():
A third-party library that simplifies HTTP requests. It provides features like automatic JSON parsing, request/response interception, and better error handling.

Question 4: What is Firebase? What features does Firebase offer?

Firebase is a platform developed by Google that provides tools and services for building web and mobile applications. It simplifies backend development, enabling developers to focus on the frontend and business logic.

Key Features of Firebase:

Authentication: Supports various methods (email/password, phone number, social logins, etc.).
Realtime Database: A NoSQL cloud database for storing and syncing data in real-time.
Cloud Firestore: A scalable, flexible NoSQL database for more complex apps.
Cloud Functions: Serverless functions triggered by Firebase events.
Hosting: Fast and secure static web hosting.
Storage: Cloud storage for file uploads and sharing.
Cloud Messaging: Free push notification service.
Analytics: Provides insights into user behavior and app usage.

Question 5: Discuss the importance of handling errors and loading states when working with APIs in React.

Handling errors and loading states is essential for providing a good user experience when interacting with APIs in React.

A) Importance of Handling Errors:

User Feedback:

Display meaningful error messages to inform users about issues (e.g., network failure or server errors).

Debugging:

Helps developers identify and fix issues faster.

Graceful Degradation:

Prevents app crashes and provides fallback mechanisms (e.g., retrying requests or showing default data).

B) Importance of Handling Loading States:

Visual Feedback:

Indicates to users that a process is ongoing, preventing them from thinking the app is unresponsive.

Smooth User Experience:

Ensures users are not interacting with incomplete or inconsistent data.

Example: Error and Loading State Handling in React:

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

const App = () => {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    axios
      .get("http://localhost:5000/users")
      .then(response => {
        setUsers(response.data);
        setLoading(false);
      })
      .catch(error => {
        setError("Failed to fetch data. Please try again.");
        setLoading(false);
      });
  }, []);

  if (loading) return <p>Loading...</p>;
  if (error) return <p>{error}</p>;

  return (
    <div>
      <h1>Users</h1>
      <ul>
        {users.map(user => (
          <li key={user.id}>{user.name}</li>
        ))}
      </ul>
    </div>
  );
};

export default App;
```

LAB EXERCISE

Task 1: o Create a React component that fetches data from a public API (e.g., a list of users) and displays it in a table format. o Create a React app with Json-server and use Get , Post , Put , Delete & patch method on Json-server API.

Part 1: React Component Fetching Data from a Public API

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


const UserTable = () => {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);


  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then((response) => {
        if (!response.ok) {
          throw new Error("Failed to fetch data");
        }
        return response.json();
      })
      .then((data) => {
        setUsers(data);
        setLoading(false);
      })
      .catch((err) => {
        setError(err.message);
        setLoading(false);
      });
  }, []);
```

```
  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error: {error}</p>;

  return (
    <div>
      <h1>User List</h1>
      <table border="1">
        <thead>
          <tr>
            <th>ID</th>
            <th>Name</th>
            <th>Email</th>
            <th>Username</th>
          </tr>
        </thead>
        <tbody>
          {users.map((user) => (
            <tr key={user.id}>
              <td>{user.id}</td>
              <td>{user.name}</td>
              <td>{user.email}</td>
              <td>{user.username}</td>
            </tr>
          ))}
        </tbody>
      </table>
    </div>
  );
};

export default UserTable;
```

Part 2: React App with JSON-Server for CRUD Operations

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

const App = () => {
  const [users, setUsers] = useState([]);
  const [newUser, setNewUser] = useState({ name: "", email: "" });
  const [editingUser, setEditingUser] = useState(null);
  const API_URL = "http://localhost:5000/users";
  // Fetch users
  const fetchUsers = async () => {
    try {
      const response = await axios.get(API_URL);
      setUsers(response.data);
    } catch (error) {
      console.error("Error fetching users:", error);
    }
  };
```

```
useEffect(() => {
  fetchUsers();
}, []);

// Add user
const addUser = async () => {
  try {
    const response = await axios.post(API_URL, newUser);
    setUsers([...users, response.data]);
    setNewUser({ name: "", email: "" });
  } catch (error) {
    console.error("Error adding user:", error);
  }
};

// Update user (PUT)
const updateUser = async (userId) => {
  try {
    const response = await axios.put(`${API_URL}/${userId}`, editingUser);
    setUsers(users.map((user) => (user.id === userId ? response.data : user)));
    setEditingUser(null);
  } catch (error) {
    console.error("Error updating user:", error);
  }
};

// Update user (PATCH)
const patchUser = async (userId, updatedField) => {
  try {
    const response = await axios.patch(`${API_URL}/${userId}`, updatedField);
    setUsers(users.map((user) => (user.id === userId ? response.data : user)));
  } catch (error) {
    console.error("Error patching user:", error);
  }
};

// Delete user
const deleteUser = async (userId) => {
  try {
    await axios.delete(`${API_URL}/${userId}`);
    setUsers(users.filter((user) => user.id !== userId));
  } catch (error) {
    console.error("Error deleting user:", error);
  }
};

return (
  <div>
    <h1>JSON-Server CRUD Operations</h1>
    <h2>Add User</h2>
    <input
```

```
 type="text"
 placeholder="Name"
 value={newUser.name}
 onChange={(e) => setNewUser({ ...newUser, name: e.target.value })}
/>
<input
 type="email"
 placeholder="Email"
 value={newUser.email}
 onChange={(e) => setNewUser({ ...newUser, email: e.target.value })}
/>
<button onClick={addUser}>Add User</button>

<h2>User List</h2>
<table border="1">
 <thead>
  <tr>
   <th>ID</th>
   <th>Name</th>
   <th>Email</th>
   <th>Actions</th>
  </tr>
 </thead>
 <tbody>
  {users.map((user) => (
   <tr key={user.id}>
    <td>{user.id}</td>
    <td>{user.name}</td>
    <td>{user.email}</td>
    <td>
     <button onClick={() => setEditingUser(user)}>Edit</button>
     <button onClick={() => deleteUser(user.id)}>Delete</button>
    </td>
   </tr>
  ))}
 </tbody>
</table>

{editingUser && (
 <div>
  <h2>Edit User</h2>
  <input
   type="text"
   value={editingUser.name}
   o
```

```
nChange={(e) => setEditingUser({ ...editingUser, name: e.target.value })}
        />
        <input
          type="email"
          value={editingUser.email}
          onChange={(e) => setEditingUser({ ...editingUser, email: e.target.value })}
        />
        <button onClick={() => updateUser(editingUser.id)}>Save</button>
        <button onClick={() => patchUser(editingUser.id, { name: "Patched Name" })}>
          Patch Name
        </button>
      </div>
    )}
  </div>
 );
};

export default App;
```

Task 2: Create a React app crud and Authentication with firebase API. o Implement google Authentication with firebase API.

Steps to Implement CRUD and Authentication Using Firebase

1. Setup Firebase

Create a Firebase Project:

Go to Firebase Console.

Create a new project and set up the web app.

Enable Firebase Authentication:

Go to the **Authentication** section.

Enable **Email/Password** and **Google** sign-in providers.

Create Firestore Database:

Navigate to the **Firestore Database** section.

Set up a new database in **test mode** for development.

Get Firebase Config:

```
  {
 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"
}
```

2) Setup Firebase in Your App

```
 // src/firebase.js
import { initializeApp } from "firebase/app";
import { getAuth, GoogleAuthProvider } from "firebase/auth";
import { getFirestore } from "firebase/firestore";

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",
};

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

3). Implement Authentication

Google Sign-In:

```
// src/components/Auth.js
import React from "react";
import { auth, provider } from "../firebase";
import { signInWithPopup, signOut } from "firebase/auth";

const Auth = ({ setUser }) => {
 const signInWithGoogle = async () => {
  try {
   const result = await signInWithPopup(auth, provider);
   setUser(result.user);
  } catch (error) {
   console.error("Error signing in:", error);
  }
 };

 const logOut = async () => {
  try {
   await signOut(auth);
   setUser(null);
  } catch (error) {
   console.error("Error signing out:", error);
  }
 };

 return (
  <div>
   <button onClick={signInWithGoogle}>Sign In with Google</button>
   <button onClick={logOut}>Log Out</button>
  </div>
 );
};

export default Auth;
```

4). Implement Firestore CRUD Operations

```
// src/components/CRUD.js
import React, { useState, useEffect } from "react";
import { db } from "../firebase";
import {
 collection,
 addDoc,
 getDocs,
 updateDoc,
 deleteDoc,
 doc,
} from "firebase/firestore";

const CRUD = () => {
 const [data, setData] = useState
```

```
([]);
 const [newItem, setNewItem] = useState("");
 const [editingId, setEditingId] = useState(null);
 const collectionRef = collection(db, "items");

 // Fetch data
 const fetchData = async () => {
  const querySnapshot = await getDocs(collectionRef);
  setData(querySnapshot.docs.map((doc) => ({ id: doc.id, ...doc.data() })));
 };

 useEffect(() => {
  fetchData();
 }, []);

 // Add item
 const addItem = async () => {
  if (newItem.trim()) {
   await addDoc(collectionRef, { name: newItem });
   setNewItem("");
   fetchData();
  }
 };

 // Update item
 const updateItem = async (id, name) => {
  const itemRef = doc(db, "items", id);
  await updateDoc(itemRef, { name });
  fetchData();
  setEditingId(null);
 };

 // Delete item
 const deleteItem = async (id) => {
  const itemRef = doc(db, "items", id);
  await deleteDoc(itemRef);
  fetchData();
 };

 return (
  <div>
   <h1>Firestore CRUD</h1>
   <div>
    <input
     type="text"
     placeholder="Add new item"
     value={newItem}
```

```
        onChange={(e) => setNewItem(e.target.value)}
      />
      <button onClick={addItem}>Add</button>
    </div>
    <ul>
      {data.map((item) => (
       <li key={item.id}>
        {editingId === item.id ? (
         <input
          type="text"
          defaultValue={item.name}
          onBlur={(e) => updateItem(item.id, e.target.value)}
         />
        ) : (
         <span onClick={() => setEditingId(item.id)}>{item.name}</span>
        )}
        <button onClick={() => deleteItem(item.id)}>Delete</button>
       </li>
      ))}
    </ul>
   </div>
  );
};

export default CRUD;
```

5) . Main App Component

```
// src/App.js
import React, { useState } from "react";
import Auth from "./components/Auth";
import CRUD from "./components/CRUD";

const App = () => {
 const [user, setUser] = useState(null);

 return (
  <div>
   {!user ? (
    <Auth setUser={setUser} />
   ) : (
    <div>
     <h2>Welcome, {user.displayName}</h2>
     <CRUD />
    </div>
   )}
  </div>
 );
};

export default App;
```

Task 3: o Implement error handling and loading states for the API call. Display a lo

1) Spinner Component

```
// src/components/Spinner.js
import React from "react";

const Spinner = () => (
  <div className="spinner">
    <style>
      {`
       .spinner {
         border: 8px solid #f3f3f3;
         border-top: 8px solid #3498db;
         border-radius: 50%;
         width: 50px;
         height: 50px;
         animation: spin 1s linear infinite;
       }

       @keyframes spin {
         0% { transform: rotate(0deg); }
         100% { transform: rotate(360deg); }
       }
      `}
    </style>
  </div>
);

export default Spinner;
```

2) Main Component with Error Handling

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

const App = () => {
  const [data, setData] = useState([]);
  const [isLoading, setIsLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      setIsLoading(true);
      setError(null); // Reset error before a new fetch

      try {
        const response = await fetch("https://jsonplaceholder.typicode.com/users");
        if (!response.ok) {
          throw new Error(`HTTP error! status: ${response.status
```

```
}`);
      }
      const result = await response.json();
      setData(result);
    } catch (err) {
      setError(err.message);
    } finally {
      setIsLoading(false); // Stop loading when fetch completes
    }
  };

  fetchData();
}, []);

if (isLoading) {
  return <Spinner />;
}

if (error) {
  return <div>Error: {error}</div>;
}

return (
  <div>
    <h1>User List</h1>
    <table border="1" style={{ width: "100%", textAlign: "left" }}>
      <thead>
        <tr>
          <th>ID</th>
          <th>Name</th>
          <th>Email</th>
          <th>Username</th>
        </tr>
      </thead>
      <tbody>
        {data.map((user) => (
          <tr key={user.id}>
            <td>{user.id}</td>
            <td>{user.name}</td>
            <td>{user.email}</td>
            <td>{user.username}</td>
          </tr>
        ))}
      </tbody>
    </table>
  </div>
);
};

export default App;
```