Experiment 3

Student Name: Saurabh Kumar **UID: 23BCS10675**
Branch: CSE **Section/Group: KRG-3-B**
Semester: 6th **Date of Performance:02/02/2026**
Subject Name: Full Stack Development – II **Subject Code: 23CSH-309**

1. Aim:

To implement centralized state management in the EcoTrack application using Redux Toolkit and to handle asynchronous data operations using Redux async thunks with proper loading and error states..

2. Objective:

1. Configure a Redux store in a React application using Redux Toolkit
2. Create and integrate Redux slices for managing application data
3. Implement asynchronous actions using Redux async thunks
4. Manage loading, success, and error states during asynchronous operations
5. Connect React components to Redux state using React-Redux hooks
6. Trigger asynchronous data fetching through Redux actions from UI components
7. Use Redux state to derive filtered views without modifying the global store
8. Enhance user experience by handling refresh actions and improving async UI feedback

3. Implementation / Code:

Implementation Description:

- The EcoTrack application implements centralized state management using Redux Toolkit, ensuring a predictable data flow and easier debugging.
- A Redux store is configured (configureStore) to manage the global state, integrating specific slices like logsSlice for handling domain-specific data.
- Asynchronous actions are handled using Redux Async Thunks (createAsyncThunk), which manage the lifecycle of API requests (pending, fulfilled, rejected) and update the state accordingly.
- React-Redux hooks (useDispatch, useSelector) are used to connect React components to the Redux store, allowing them to trigger data fetching and subscribe to state updates.

- Selectors are implemented to encapsulate state lookup logic and derive filtered views (e.g., high carbon footprint logs) directly from the Redux state without modifying the underlying data, enhancing performance and maintainability.

- The application manages loading and error states within the Redux slice, enabling the UI to provide immediate and accurate feedback to the user during data operations.

Sample Code Snippet:

src > store > store.js > store

```
1  import { configureStore } from "@reduxjs/toolkit";
2  import logsReducer from "./logSlice";
3
4
5  export const store = configureStore({
6      reducer:{
7          logs:logsReducer,
8      }
9  })
```

src > store > logSlice.js > logsSlice > extraReducers > addCase() callback

```
1   import { createSlice,createAsyncThunk } from "@reduxjs/toolkit";
2
3   export const fetchLogs = createAsyncThunk(
4       "logs/fetchLogs",
5       async()=>{
6           await new Promise((resolve)=>setTimeout(resolve,1000));
7
8           return [
9     { id: 1, activity: "Car Travel", carbon: 4 },
10    { id: 2, activity: "Electricity Usage", carbon: 7 },
11    { id: 3, activity: "Cycling", carbon: 0 },
12  ];
13      }
14  )
15
16  const logsSlice = createSlice({
17      name:"logs",
18      initialState:{
19          data :[],
20          status:"idle",
21          error:null,
22      },
23      reducers:{},
24      extraReducers:(builder)=>{
25          builder.addCase(fetchLogs.pending,(state)=>{
26              state.status="loading";
27          })
28          .addCase(fetchLogs.fulfilled,(state,action)=>{
29              state.status="succeeded";
30              state.data=action.payload;
31          })
32          .addCase(fetchLogs.rejected,(state,action)=>{
33              state.status="failed";
34              state.error=action.error.message;
35          })
36
37
38
39      }
40      })
41
42
43  export default logsSlice.reducer;
```

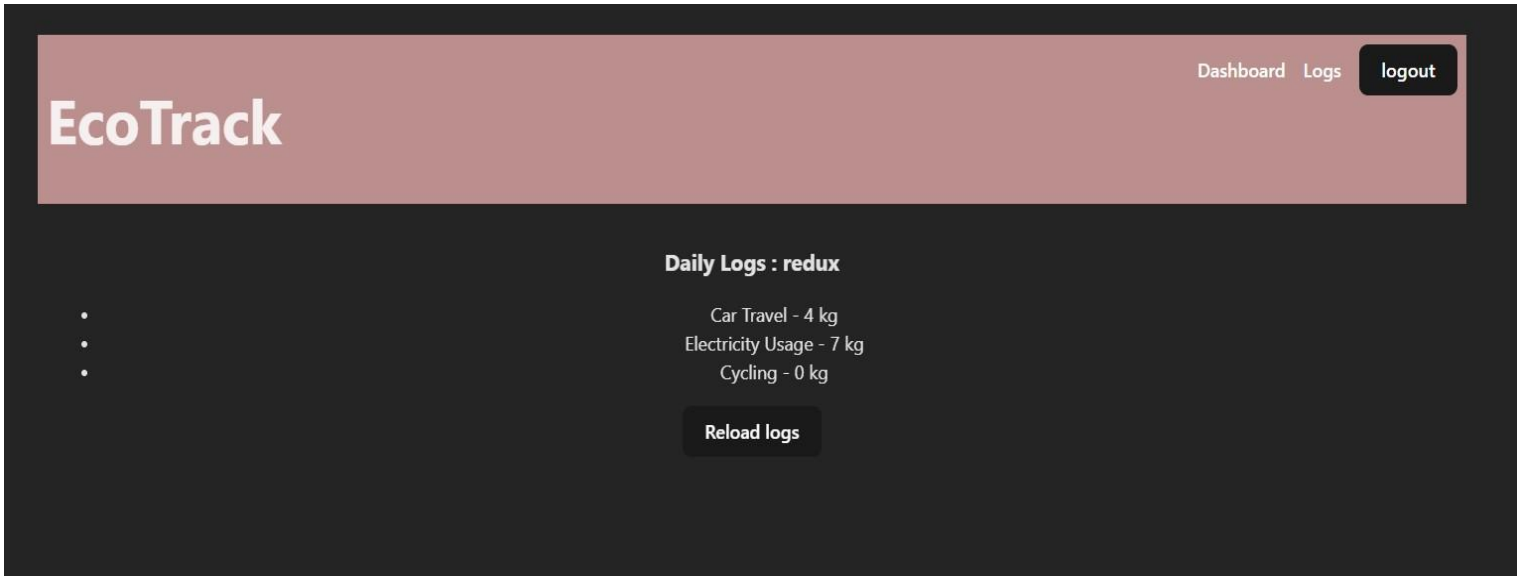
src > pages > Logs.jsx > Logs

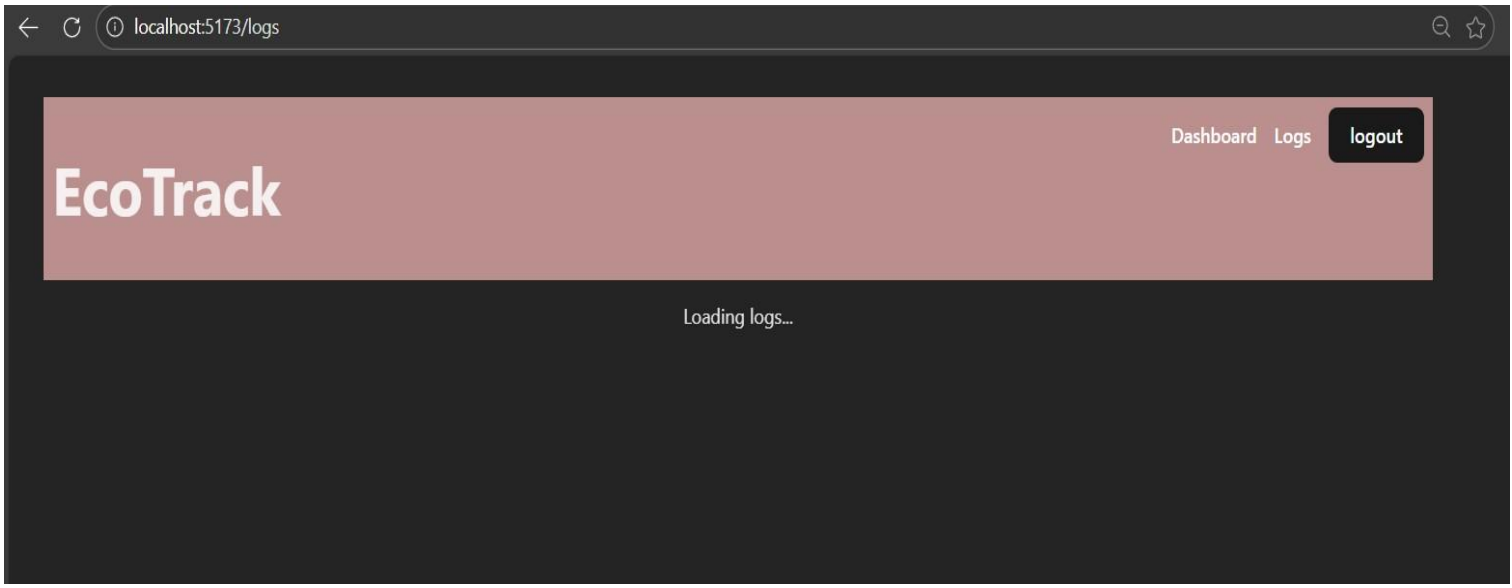
```jsx
import { useEffect } from "react";
import { useDispatch,useSelector } from "react-redux";
import { fetchLogs } from "../store/logSlice";


const Logs = () =>{

    const dispatch = useDispatch();
    const {data,status,error} = useSelector((state)=>state.logs);

    useEffect(()=>{
       if(status === "idle"){
        dispatch(fetchLogs());
       }
    },[status,dispatch]);

    const handleRefresh = ()=>{
        dispatch(fetchLogs());
    }



    if(status === "loading") {
        return <p>Loading logs...</p>
    }

    return(
        <div style={{padding:"1rem"}}>

            <h3>Daily Logs : redux</h3>
            <ul>
                {data.map((log)=>(
                    <li key={log.id}>{log.activity} - {log.carbon} kg</li>
                ))}
            </ul>

            <button onClick={handleRefresh} >Reload logs</button>


        </div>


    )

}

export default Logs;
```

4. Output:

- Successfully implemented centralized state management using Redux Toolkit.

- Integrated async thunks to handle API calls with proper loading and error states.
- Connected UI components to Redux using hooks for seamless state updates.
- Improved user experience with responsive async feedback and refresh actions.





5. Learning Outcomes

- Learned how to configure a Redux store and create feature-based slices.
- Understood the use of async thunks for managing asynchronous operations.
- Gained experience in handling loading, success, and error states in Redux.
- Learned to derive filtered views using selectors without mutating global state.