Experiment No. 3

Aim :-
Manage complex state with Redux or Context API

Prerequisites :-
Before starting this project, you should have:

1. **Basic knowledge of React**
    - Components, props, state, hooks (useState, useEffect).
2. **Familiarity with JavaScript**
    - Arrow functions, array methods (map, filter), object destructuring.
3. **Understanding of Redux basics**
    - Store, reducer, action, dispatch, useSelector, useDispatch.
4. **Tailwind CSS basics**
    - Classes for styling, spacing, colors.
5. **Node.js and npm installed**
    - To run React and install dependencies (react-redux, @reduxjs/toolkit).

Theory :-
a) Problem Statement

In a typical task management app, multiple components need to **access and update the same task list**. Passing state via props can get messy as the app grows. Redux provides a **centralized state management solution**, making state predictable and easier to manage.

b) Redux Core Concepts

1. **Store** – Central place that holds the state of the app.
2. **State** – The data itself (task list in our case).
3. **Action** – Object that describes what happened (addTask, deleteTask, toggleTask).
4. **Reducer** – Pure function that **updates the state** based on actions.
5. **Dispatch** – Function used to **send actions** to the store.
6. **Selector** – Function to **read state from the store** (useSelector hook).

c) Flow in Task Manager

1. User adds a task → dispatches addTask action → reducer updates store → component re-renders.
2. User toggles or deletes a task → dispatches toggleTask or deleteTask → reducer updates store → UI updates automatically.
3. Tasks are styled using **Tailwind CSS** for responsive, clean UI.

Program Code :-

- redux/store.js :-

```
import { configureStore } from "@reduxjs/toolkit";
import tasksReducer from "./tasksSlice";

export const store = configureStore({
  reducer: {
    tasks: tasksReducer,
  },
});

```

- redux/tasksSlice.js :-

```
import { createSlice, createAsyncThunk, createSelector } from "@reduxjs/toolkit";

// 1️⃣ Async thunk to simulate fetching tasks from API
export const fetchTasks = createAsyncThunk(
  "tasks/fetchTasks",
  async () => {
    // Simulate API delay
    return new Promise((resolve) => {
      setTimeout(() => {
        resolve([
          { id: 1, text: "Learn Redux", completed: false },
          { id: 2, text: "Build Task App", completed: true },
        ]);
      }, 1000);
    });
  }
);

// Load tasks from localStorage
const loadFromLocalStorage = () => {
  try {
    const serializedState = localStorage.getItem("tasks");
    if (serializedState === null) return [];
    return JSON.parse(serializedState);
  } catch (e) {
    return [];
  }
};

// Save tasks to localStorage
const saveToLocalStorage = (tasks) => {
  localStorage.setItem("tasks", JSON.stringify(tasks));
};

const tasksSlice = createSlice({
  name: "tasks",
  initialState: {
    items: loadFromLocalStorage(),
```

```
    status: "idle", // idle | loading | succeeded | failed
  },
  reducers: {
    addTask: (state, action) => {
      state.items.push({ id: Date.now(), text: action.payload, completed: false });
      saveToLocalStorage(state.items);
    },
    toggleTask: (state, action) => {
      const task = state.items.find((task) => task.id === action.payload);
      if (task) task.completed = !task.completed;
      saveToLocalStorage(state.items);
    },
    deleteTask: (state, action) => {
      state.items = state.items.filter((task) => task.id !== action.payload);
      saveToLocalStorage(state.items);
    },
  },
  extraReducers: (builder) => {
    builder
      .addCase(fetchTasks.pending, (state) => {
        state.status = "loading";
      })
      .addCase(fetchTasks.fulfilled, (state, action) => {
        state.status = "succeeded";
        state.items = action.payload;
        saveToLocalStorage(state.items);
      })
      .addCase(fetchTasks.rejected, (state) => {
        state.status = "failed";
      });
  },
});


export const { addTask, toggleTask, deleteTask } = tasksSlice.actions;
export default tasksSlice.reducer;
```

- TaskInput.js :-

```jsx
import React, { useState } from "react";
import { useDispatch } from "react-redux";
import { addTask } from "../redux/tasksSlice";

const TaskInput = () => {
  const [text, setText] = useState("");
  const dispatch = useDispatch();

  const handleAdd = () => {
    if (text.trim() === "") return;
    dispatch(addTask(text));
    setText("");
  };

  return (
    <div className="flex mb-4">
      <input
        type="text"
        value={text}
        onChange={(e) => setText(e.target.value)}
        className="border border-gray-300 rounded-l px-4 py-2 flex-1"
        placeholder="Add a task"
      />
      <button
        onClick={handleAdd}
        className="bg-blue-500 text-white px-4 py-2 rounded-r hover:bg-blue-600"
      >
        Add
      </button>
    </div>
  );
};

export default TaskInput;
```

- TaskItem.js :-

```jsx
import React from "react";
import { useDispatch } from "react-redux";
import { toggleTask, deleteTask } from "../redux/tasksSlice";

const TaskItem = ({ task }) => {
  const dispatch = useDispatch();

  return (
    <li className="flex justify-between items-center bg-gray-100 p-2 mb-2 rounded">
      <span
        onClick={() => dispatch(toggleTask(task.id))}
        className={`${task.completed ? "line-through text-gray-400" : ""} cursor-pointer`}
      >
        {task.text}
      </span>
      <button
        onClick={() => dispatch(deleteTask(task.id))}
        className="bg-red-500 text-white px-2 py-1 rounded hover:bg-red-600"
      >
        Delete
      </button>
    </li>
  );
};

export default TaskItem;
```

- TaskList.js :-

```jsx
import React from "react";
import { useSelector } from "react-redux";
import TaskItem from "./TaskItem";

const TaskList = () => {
  const tasks = useSelector((state) => state.tasks.items);

  if (tasks.length === 0) return <p className="text-gray-500">No tasks yet!</p>;

  return (
    <ul>
      {tasks.map((task) => (
        <TaskItem key={task.id} task={task} />
      ))}
    </ul>
  );
};

export default TaskList;
```

- App.js :-

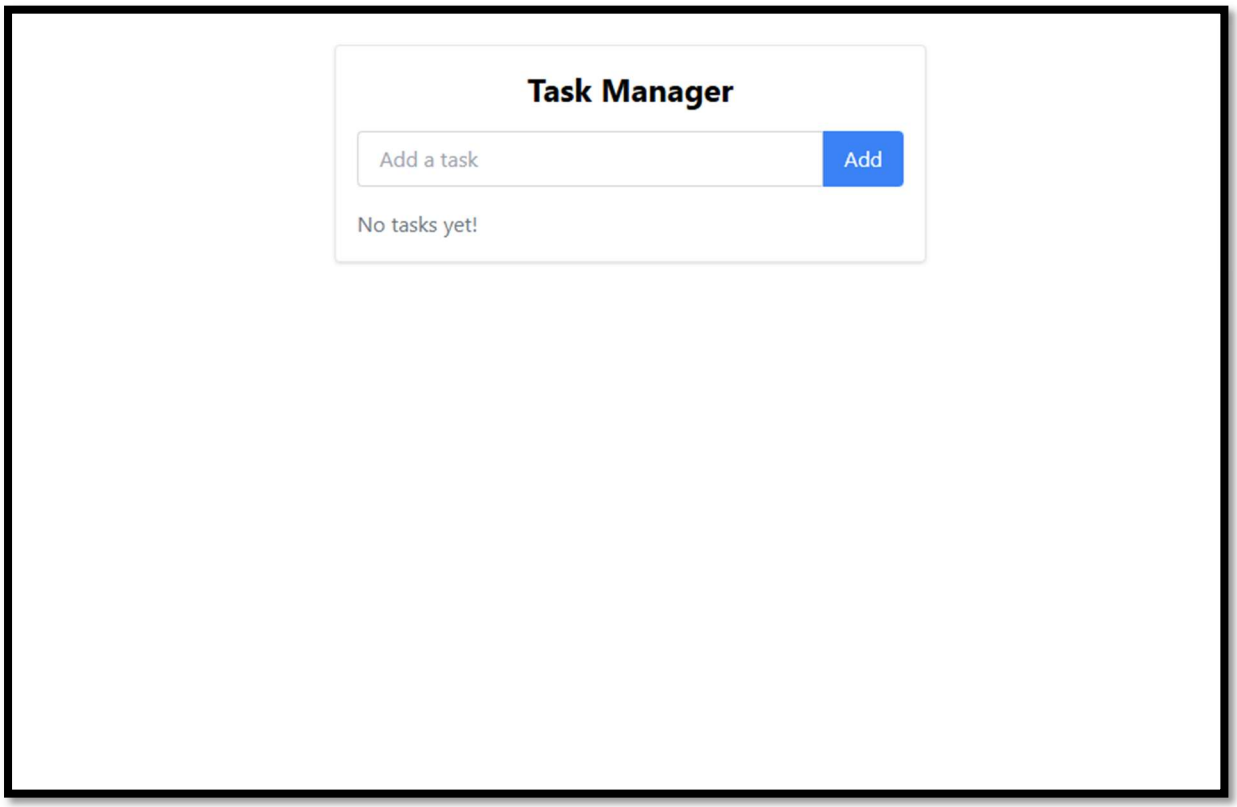
```jsx
import React, { useEffect } from "react";
import TaskInput from "./components/TaskInput";
import TaskList from "./components/TaskList";
import { useDispatch, useSelector } from "react-redux";
import { fetchTasks } from "./redux/tasksSlice";

function App() {
  const dispatch = useDispatch();
  const status = useSelector((state) => state.tasks.status);

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

  return (
    <div className="max-w-md mx-auto mt-10 p-4 border rounded shadow">
      <h1 className="text-2xl font-bold mb-4 text-center">Task Manager</h1>
      <TaskInput />
      {status === "loading" ? (
        <p className="text-center text-gray-500">Loading tasks...</p>
      ) : (
        <TaskList />
      )}
    </div>
  );
}

export default App;
```

- Index.js :-

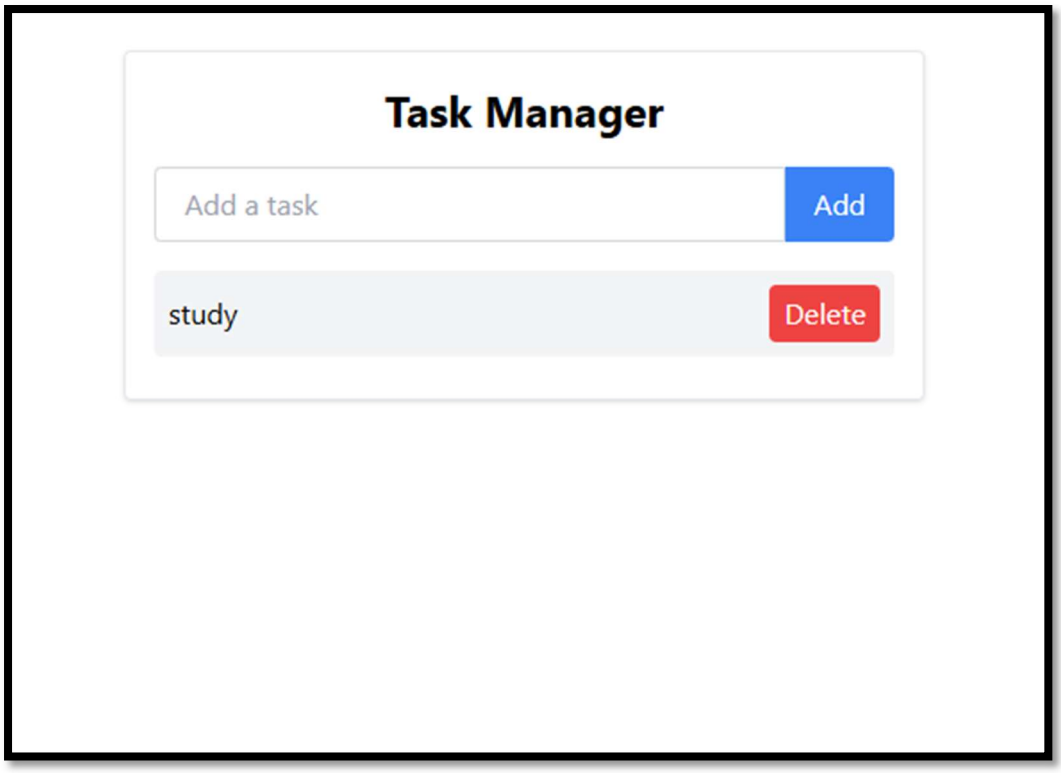
```jsx
import React from "react";
import ReactDOM from "react-dom/client";
import "./index.css";
import App from "./App";
import { store } from "./redux/store";
import { Provider } from "react-redux";

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

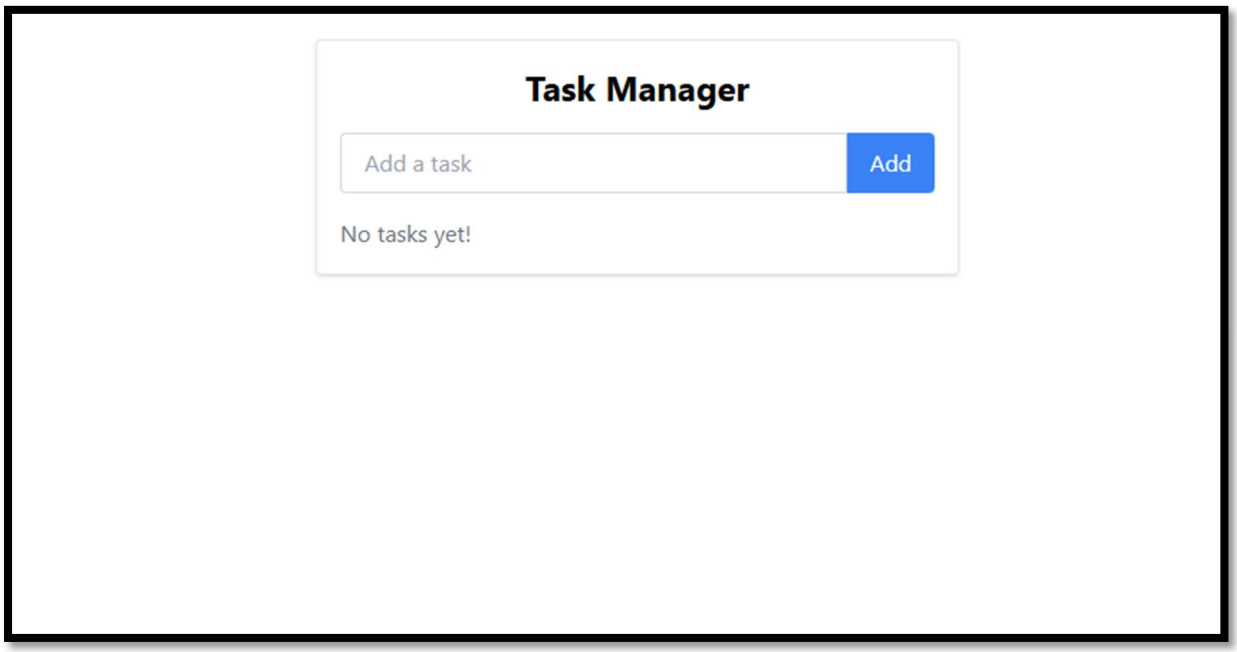
Output :-

- HomePage :-

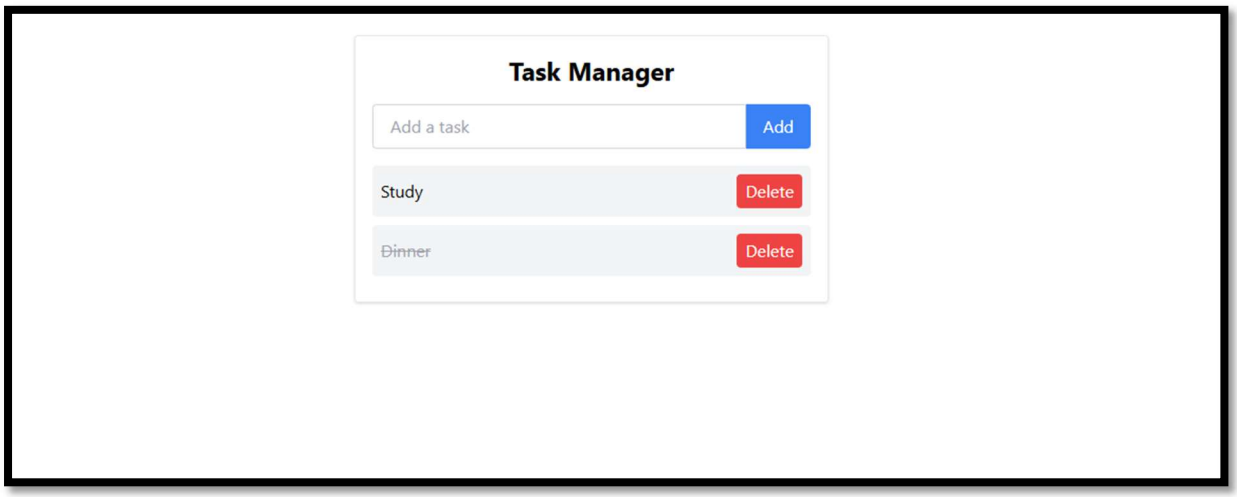


- Adding new task (Using redux add and delete tasks) :-





- After refreshing the webpage (Use of LocalStorage Persistence) :-



30% Extra Content :-

Redux Thunk (Async Actions) :-

- **Purpose:** Redux Thunk allows Redux to handle **asynchronous operations** like API calls.
- **How it works:**
  - Normally, Redux reducers are synchronous—they just take the current state and an action and return a new state.
  - Redux Thunk lets you **dispatch functions instead of plain actions**. These functions can perform async tasks (e.g., fetching data) and then dispatch normal actions when completed.
- **Example in Task Manager:**

    - fetchTasks() simulates fetching tasks from a server.
    - While waiting for the "API response," the app can show a **loading indicator**.
    - When the data is returned, it is automatically saved to the Redux store and displayed in the UI.

Benefit: Makes the app feel **more realistic**, as if it is connected to a backend, and prepares it for real-world API integration.

LocalStorage Persistence :-

- **Purpose:** LocalStorage persistence ensures that **tasks remain saved even if the user reloads the page or closes the browser**.
- **How it works:**
    - When the app loads, the Redux store initializes tasks from LocalStorage.
    - After any state-changing action (addTask, toggleTask, deleteTask), the updated tasks are **saved back to LocalStorage**.
- **Example in Task Manager:**
    - Adding a new task automatically updates LocalStorage.
    - Reloading the page retrieves tasks from LocalStorage, so the user sees the same list as before.

Benefit: Improves **user experience** by giving the app **persistent state** without needing a backend database.

Conclusion :-

The **Task Manager project** demonstrates how Redux can **manage complex state** in a React application efficiently:

- All tasks are **centralized in the Redux store**, avoiding prop-drilling.
- Components automatically update when state changes.
- Advanced features like **Redux Thunk** and **LocalStorage persistence** make the app **more realistic and production-ready**.
- Styling with Tailwind CSS ensures the UI is clean and responsive