AeroAspire

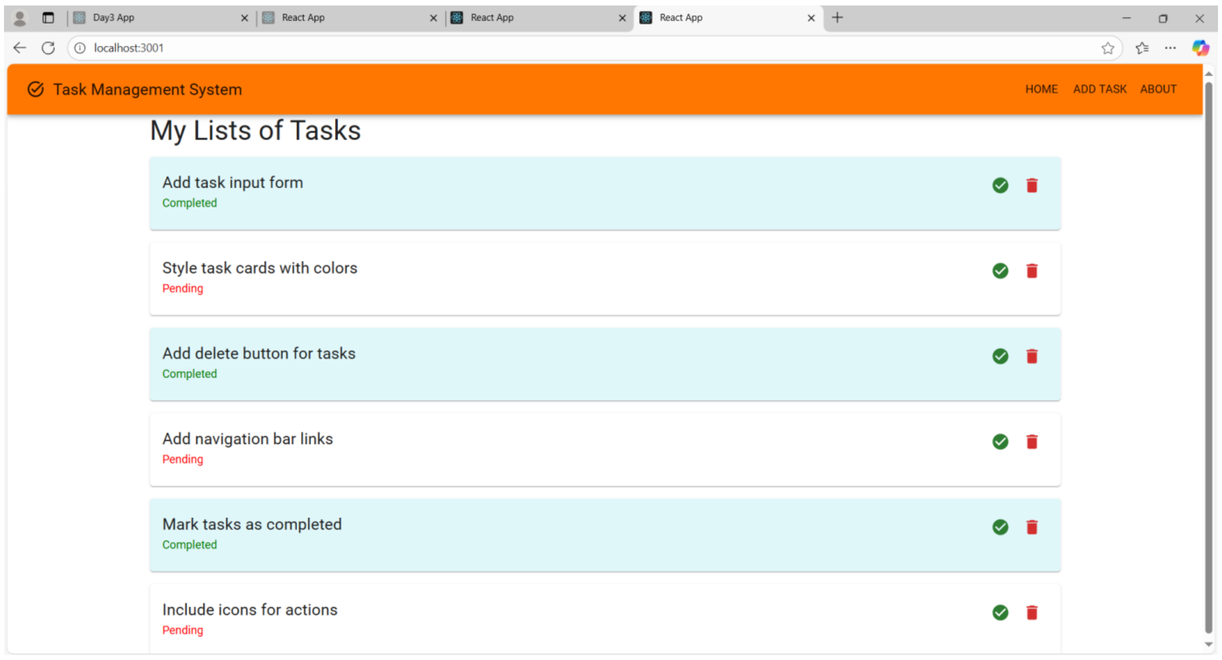
SDE Intern

Goutham V

Week 2 – Day4 (03rd October)

Task:

Add routing: Home, Add Task, About; navigation bar, Save tasks to localStorage; load on mount; polish UI; icons; theming



My Lists of Tasks

Add task input form
Completed

Style task cards with colors
Pending

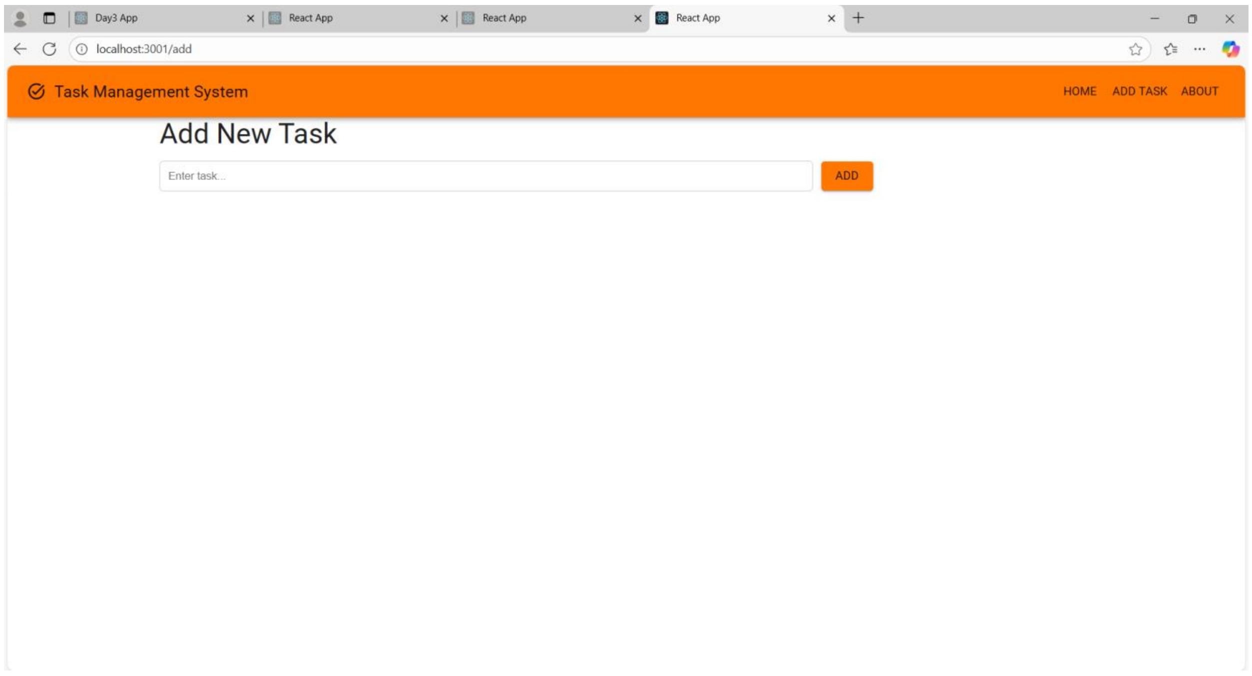
Add delete button for tasks
Completed

Add navigation bar links
Pending

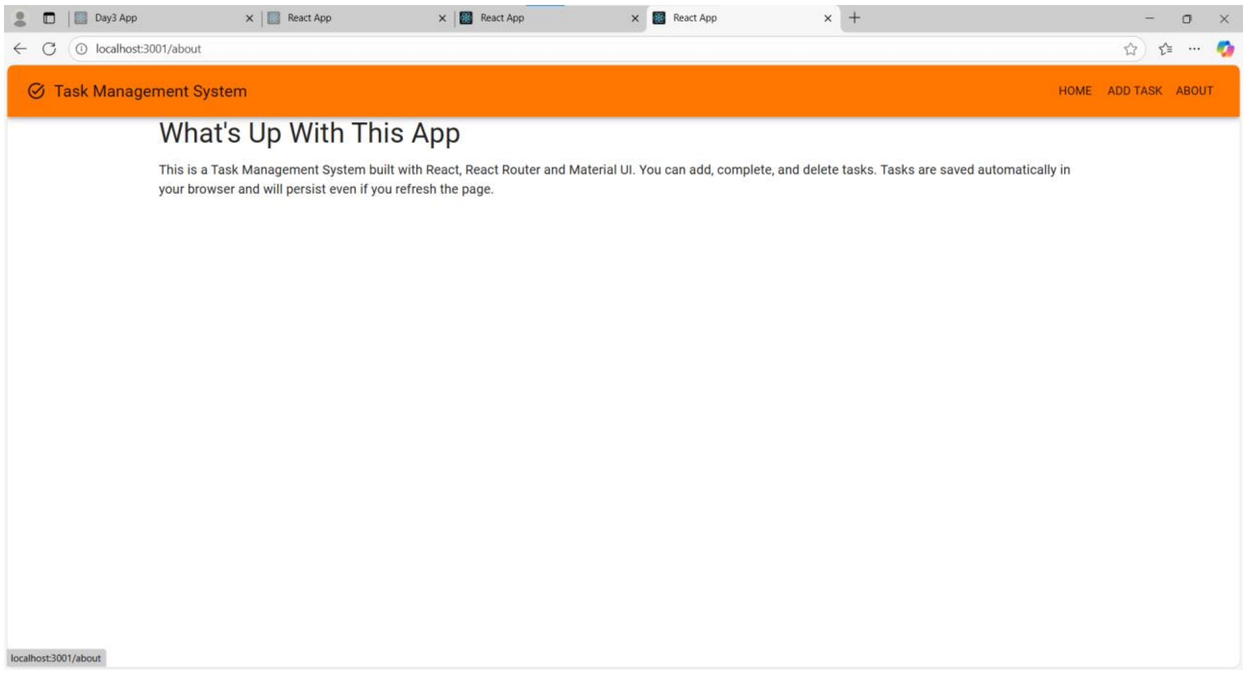
Mark tasks as completed
Completed

Include icons for actions
Pending

This is the home page showing all tasks with complete and delete actions.



Here you can add a new task using the input form.



What's Up With This App

This is a Task Management System built with React, React Router and Material UI. You can add, complete, and delete tasks. Tasks are saved automatically in your browser and will persist even if you refresh the page.

This page gives a brief overview of the Task Management System.

Steps I Followed

Day 4 – Task Management System

1. Created a new React project for Day 4 inside my AeroAspire folder.
2. Installed all the necessary packages:
   - Material UI (@mui/material, @emotion/react, @emotion/styled)
   - MUI icons (@mui/icons-material)
   - React Router (react-router-dom)
3. Set up *BrowserRouter* in index.js to enable routing between pages.
4. Created App.js with:
   - A *navigation bar* with Home, Add Task, About links
   - *Routing* using Routes and Route
   - *Tasks state* using useState
   - *Add, complete, and delete task functionality*
5. Added *useEffect* to save tasks to localStorage whenever tasks changed.
6. Added *useEffect* to load tasks from localStorage on app mount.
7. Built *Home page* showing all tasks in *cards* with clear visual distinction for completed and pending tasks.
8. Built *Add Task page* with a form to add new tasks.
9. Built *About page* describing the Task Management System.
10. Styled the UI using *Material UI components*: AppBar, Toolbar, Card, Button, Typography, Stack.
11. Added *icons* for actions:
    - Checkmark icon for completing a task
    - Trash icon for deleting a task
12. Applied *custom theme* for primary and secondary colors.
13. Tested the app in the browser:
    - Checked navigation works
    - Added tasks
    - Completed tasks
    - Deleted tasks
14. Added *dummy tasks* to see how cards look with completed and pending tasks.
15. Took screenshots of:
    - Home page
    - Add Task page
    - About page

Reflection,

1. Describe how client-side routing works (history API or hash routing)

Client-side routing means the ***browser doesn’t reload the whole page*** when you navigate. Instead, React changes what’s shown on the screen by updating the *URL in the browser* using either:

- **History API** → changes the URL path like /add or /about without reloading. This is what React Router normally uses.
- **Hash routing** → changes the URL hash like /#/add for older browsers.

React Router listens to these changes and decides which component to render.

2. What happens when you navigate: how React Router matches route and renders components

When I click a link or go to a URL, React Router:

1. Checks all the defined <Route> paths in my Routes.
2. Finds the first path that matches the current URL.
3. Renders the ***component assigned to that path*** without refreshing the page.

For example, /add renders the AddTask component, / renders the Home component.

3. How to pass params or query params; nested routes

- **Params**: You can define a route like /task/:id and access the id in the component with useParams().
- **Query params**: You can use useLocation() to get the URL search string like ?filter=pending and parse it.
- **Nested routes**: You can render a child route inside a parent route. For example, /projects can have /projects/:id inside it.

This is useful when you want *dynamic pages or pages inside pages*.

4. What is the flow: writing to localStorage → reading on app startup?

The flow I followed is simple:

1. When a task is added, completed, or deleted, I *update the **tasks** state**.*
2. I immediately save the new state to ***localStorage*** using localStorage.setItem("tasks", JSON.stringify(tasks)).
3. On app startup, I *read the stored tasks* from localStorage using localStorage.getItem("tasks").
4. If there are saved tasks, I convert the string back to an array with JSON.parse() and set it as the initial state.

This ensures *tasks appear even after refreshing the page***.**

5. How do you sync state with localStorage safely (e.g. updates, JSON parse/stringify)

- Always use ***JSON.stringify*** when saving, because localStorage can only store strings.
- Always use ***JSON.parse*** when reading to get the array/object back.
- Never mutate the state directly; always create a *new array/object***.***
- Update localStorage *inside a **useEffect*** that listens to state changes, so it automatically saves whenever tasks change.

This prevents *stale or broken data***.***

6. What performance / size concerns with storing too much in localStorage

- localStorage is *limited in size* (usually around 5MB).
- Storing *large amounts of data* can slow down reading/writing.

- Complex objects need *JSON stringify/parse*, which can also take time if the data is huge.
- It’s not suitable for *real-time updates or very large datasets*. For big apps, a *backend database* is better.