Day 4 CC 3: Coding Challenge: Build a Real-time Task Management Dashboard

Problem Statement:

You are tasked with developing a **Real-time Task Management Dashboard** for a remote team using React and Express with WebSockets. The dashboard should allow users to:

- Create, update, and delete tasks.
- Assign tasks to team members.
- Apply real-time updates when any user makes a change.
- Implement different UI themes using styled-components or Tailwind CSS.
- Use controlled components for form inputs and manage form states effectively.
- Implement a notification system that alerts users of task updates in real-time.
- Persist tasks using Express and WebSockets instead of REST API.

User Stories:

User Role: Team Member

- **Task Creation:** As a team member, I should be able to add new tasks with a title, description, deadline, and assignee so that everyone can see and track them.
- **Task Management:** As a team member, I should be able to edit or delete tasks assigned to me.
- **Real-time Updates:** As a team member, I should see updates instantly if any task is added, updated, or deleted by another user.
- **Theming:** As a user, I should be able to switch between light and dark themes for better accessibility.
- **Notifications:** As a team member, I should receive real-time alerts when a new task is assigned to me.

User Role: Admin

- **Task Overview:** As an admin, I should be able to see all tasks and their status.
- **Assign Tasks:** As an admin, I should be able to assign and reassign tasks.

- **Remove Users:** As an admin, I should be able to remove inactive users from the dashboard.

Implementation Steps:

1. Frontend (React) - Task Dashboard

Tech Stack: React, Hooks, Styled Components/Tailwind CSS, Context API/Redux

- Create **TaskList** and **TaskItem** components.
- Use **Controlled Components** for form inputs.
- Implement **useEffect** and lifecycle methods for fetching initial data and handling updates.
- Implement theme switcher using **Styled Components or Tailwind CSS.**
- Use **useContext or Redux** for state management.

2. Backend (Express & WebSockets) - Real-time Server

Tech Stack: Node.js, Express, Socket.io

- Setup an Express server with WebSockets using **Socket.io.**
- Implement events: `taskAdded`, `taskUpdated`, `taskDeleted`, and `userNotified`.
- Store tasks in an in-memory store or MongoDB (optional).
- Implement real-time task update notifications.

3. Connecting Frontend & Backend

- Use WebSockets (`socket.io-client`) in React to listen for real-time events.
- Dispatch updates to **React state (Context API/Redux).**
- Ensure smooth UI updates without reloading.

Expected Outcome:

- A **responsive real-time dashboard** with interactive task management.
- **Instant updates** across all users without manual refresh.
- A **theme-switching feature** for accessibility.
- **State management** using Context API or Redux.

- **Optimized component lifecycle** using Hooks.

💡 Hints:

1. Frontend (React)

🔷 Use Functional Components & Hooks:

- `useState` for local state (task inputs, form handling).
- `useEffect` for listening to real-time updates.
- `useContext` or **Redux** for global state (task list, notifications).

🔷 WebSockets Integration:

- Use `socket.io-client` to listen to real-time task updates from the backend.
- Implement events like `taskAdded`, `taskUpdated`, and `taskDeleted`.

🔷 Styling Options:

- Use **Tailwind CSS** for rapid styling.
- Use **Styled Components** for theme-based styling.
- Implement a **dark mode toggle** using `useState` or **CSS variables**.

---

2. Backend (Express & WebSockets)

🔷 Use Express for API & WebSockets:

- Create an **Express server** with WebSocket (`socket.io`) support.
- Store tasks in an **in-memory store** or **MongoDB**.
- Emit WebSocket events (`io.emit()`) when tasks are modified.

🔷 Database Storage (Optional)

- Use **MongoDB + Mongoose** for persistent storage.
- Implement **CRUD operations** for tasks (`GET, POST, DELETE`).

🛠 Boilerplate Code

1. Backend: Express & WebSockets Setup

```
// Install dependencies: express, socket.io, cors
const express = require("express");
const http = require("http");
const { Server } = require("socket.io");
const cors = require("cors");


const app = express();
const server = http.createServer(app);
const io = new Server(server, {
  cors: { origin: "*" },
});


app.use(cors());
app.use(express.json());


// In-memory task storage
let tasks = [];


// WebSocket connection
io.on("connection", (socket) => {
  console.log("User connected:", socket.id);


  // Send initial tasks
  socket.emit("loadTasks", tasks);


  // Handle adding a task
  socket.on("addTask", (task) => {
    tasks.push(task);
    io.emit("taskUpdated", tasks); // Broadcast update
  });


  // Handle deleting a task
  socket.on("deleteTask", (taskId) => {
    tasks = tasks.filter(task => task.id !== taskId);
    io.emit("taskUpdated", tasks);
  });


  socket.on("disconnect", () => {
    console.log("User disconnected:", socket.id);
```

```
  });
});

server.listen(5000, () => console.log("Server running on port 5000"));
```

---

2. Frontend: React + WebSockets Setup

Install Dependencies

```
npx create-react-app task-dashboard
cd task-dashboard
npm install socket.io-client styled-components
```

Task Dashboard Component

```
import React, { useState, useEffect } from "react";
import io from "socket.io-client";

const socket = io("http://localhost:5000");

const TaskDashboard = () => {
  const [tasks, setTasks] = useState([]);
  const [newTask, setNewTask] = useState("");

  useEffect(() => {
    socket.on("loadTasks", (loadedTasks) => {
      setTasks(loadedTasks);
    });

    socket.on("taskUpdated", (updatedTasks) => {
      setTasks(updatedTasks);
    });

    return () => socket.disconnect();
  }, []);

  const addTask = () => {
    if (newTask.trim() !== "") {
      const task = { id: Date.now(), title: newTask };
```

```
      socket.emit("addTask", task);
      setNewTask("");
    }
  };

  const deleteTask = (id) => {
    socket.emit("deleteTask", id);
  };

  return (
    <div>
      <h1>Task Dashboard (Real-time)</h1>
      <input
        type="text"
        value={newTask}
        onChange={(e) => setNewTask(e.target.value)}
        placeholder="Enter task"
      />
      <button onClick={addTask}>Add Task</button>

      <ul>
        {tasks.map((task) => (
          <li key={task.id}>
            {task.title} <button onClick={() =>
deleteTask(task.id)}>X</button>
          </li>
        ))}
      </ul>
    </div>
  );
};

export default TaskDashboard;
```

---

3. Add Theme Toggle (Styled Components)

```
import React, { useState } from "react";
import styled, { ThemeProvider } from "styled-components";

const lightTheme = { background: "#fff", color: "#333" };
const darkTheme = { background: "#333", color: "#fff" };
```

```
const Container = styled.div`
  background: ${(props) => props.theme.background};
  color: ${(props) => props.theme.color};
  min-height: 100vh;
  padding: 20px;
`;


const ThemeToggle = () => {
  const [theme, setTheme] = useState(lightTheme);

  return (
    <ThemeProvider theme={theme}>
      <Container>
        <button onClick={() => setTheme(theme === lightTheme ?
darkTheme : lightTheme)}>
          Toggle Theme
        </button>
      </Container>
    </ThemeProvider>
  );
};


export default ThemeToggle;
```

🚀 Expected Outcome

✅ **Users can add, view, and delete tasks in real-time**
✅ **WebSocket ensures changes are reflected instantly**
✅ **Dark mode toggle improves UI experience**

📌 Next Steps (Bonus)

🔹 Add **Redux** for state management
🔹 Store tasks in a **database (MongoDB)**
🔹 Implement a **real-time notification system**