NAME: JANHVI SINGH

REGISTRATION NUMBER: 22BCE8544

Assignment 13

App.jsx

// src/App.jsx

import React from 'react';

import UserList from './UserList'; // Import the new UserList component

import './styles.css'; // Import global styles

function App() {

  return (

    <div className="App flex items-center justify-center min-h-screen bg-gray-100 dark:bg-gray-900">

      {/\* Render the UserList component \*/}

      <UserList />

    </div>

  );

}

export default App;

UserList.jsx

// src/UserList.jsx

import React, { useState, useEffect } from 'react';

// Mock API function to simulate fetching user data

const fetchUsersMockApi = () => {

  return new Promise((resolve) => {

    // Simulate network delay

    setTimeout(() => {

      resolve([

        { id: 1, name: 'Alice Smith', email: 'alice@example.com' },

        { id: 2, name: 'Bob Johnson', email: 'bob@example.com' },

        { id: 3, name: 'Charlie Brown', email: 'charlie@example.com' },

        { id: 4, name: 'Diana Prince', email: 'diana@example.com' },

        { id: 5, name: 'Eve Adams', email: 'eve@example.com' },

      ]);

    }, 2000); // 2-second delay

  });

};

const UserList = () => {

  // State for storing the list of users

  const [users, setUsers] = useState([]);

  // State for managing the loading status

  const [loading, setLoading] = useState(true);

  // State for storing any error messages

  const [error, setError] = useState(null);

  // State for the search input value

  const [searchTerm, setSearchTerm] = useState('');

  // useEffect for data fetching when the component mounts

  // The empty dependency array ensures this effect runs only once after the initial render

  useEffect(() => {

    const getUsers = async () => {

      try {

        setLoading(true); // Set loading to true before fetching

        const data = await fetchUsersMockApi(); // Call the mock API

        setUsers(data); // Update users state with fetched data

      } catch (err) {

        setError('Failed to fetch users. Please try again.'); // Set error state on failure

        console.error('Error fetching users:', err);

      } finally {

        setLoading(false); // Set loading to false after fetch completes (success or failure)

      }

    };

    getUsers(); // Execute the fetching function

    // No cleanup needed for data fetching if it's a one-time operation on mount

  }, []); // Empty dependency array means this runs once on mount

  // useEffect for the document title timer side effect and cleanup

  // This effect also runs once on mount due to the empty dependency array

  useEffect(() => {

    // Set up a timer to update the document title every 5 seconds

    const timerId = setInterval(() => {

      document.title = `Active Users: ${users.length}`;

    }, 5000); // 5000 milliseconds = 5 seconds

    // Cleanup function for the timer

    // This function runs when the component unmounts or before the effect re-runs (if dependencies change)

    return () => {

      clearInterval(timerId); // Clear the interval to prevent memory leaks

      document.title = 'React App'; // Reset document title when component unmounts

    };

  }, [users.length]); // Dependency array: re-run this effect if the number of users changes

  // Function to handle changes in the search input field

  const handleSearchChange = (event) => {

    setSearchTerm(event.target.value); // Update the search term state

  };

  // Filter users based on the search term

  const filteredUsers = users.filter(user =>

    user.name.toLowerCase().includes(searchTerm.toLowerCase())

  );

  return (

    <div className="user-list-container p-8 rounded-lg shadow-xl max-w-2xl w-full bg-white dark:bg-gray-800 text-gray-900 dark:text-gray-100 transition-colors duration-300">

      <h1 className="text-3xl font-bold mb-6 text-center">User Dashboard</h1>

      {/\* Search Input Field \*/}

      <div className="mb-6">

        <input

          type="text"

          placeholder="Search users by name..."

          value={searchTerm}

          onChange={handleSearchChange}

          className="w-full px-4 py-2 border rounded-md focus:outline-none focus:ring-2 focus:ring-blue-500 bg-gray-50 dark:bg-gray-700 dark:text-gray-100 transition-colors"

        />

      </div>

      {/\* Conditional rendering based on loading, error, or data availability \*/}

      {loading ? (

        // Loading state display

        <div className="text-center py-8">

          <p className="text-xl font-medium text-blue-600 dark:text-blue-400">Loading users...</p>

        </div>

      ) : error ? (

        // Error state display

        <div className="text-center py-8 text-red-600 dark:text-red-400">

          <p className="text-xl font-medium">{error}</p>

        </div>

      ) : (

        // Display user list if data is loaded successfully

        <div className="user-list">

          <h2 className="text-2xl font-semibold mb-4">Active Users ({filteredUsers.length})</h2>

          {filteredUsers.length === 0 ? (

            <p className="text-center text-lg text-gray-600 dark:text-gray-400">No users found matching your search.</p>

          ) : (

            <ul className="space-y-4">

              {filteredUsers.map(user => (

                <li

                  key={user.id}

                  className="p-4 bg-gray-50 dark:bg-gray-700 rounded-md shadow-sm border border-gray-200 dark:border-gray-600 flex justify-between items-center transition-colors duration-200"

                >

                  <div>

                    <p className="font-medium text-lg">{user.name}</p>

                    <p className="text-sm text-gray-500 dark:text-gray-300">{user.email}</p>

                  </div>

                  {/\* Optional: Add user-specific actions here \*/}

                </li>

              ))}

            </ul>

          )}

        </div>

      )}

      {/\* Tailwind CSS CDN for styling \*/}

      <script src="https://cdn.tailwindcss.com"></script>

    </div>

  );

};

export default UserList;

styles.css

/\* src/styles.css \*/

/\* Global styles for the body and root \*/

html, body, #root, .App {

  height: 100%; /\* Ensure full height \*/

  margin: 0;

  padding: 0;

  font-family: "Inter", sans-serif; /\* Use Inter font \*/

  box-sizing: border-box;

}

/\* Default body theme (light) \*/

body {

  background-color: #f0f2f5; /\* Light grey background \*/

  color: #333; /\* Dark text \*/

  transition: background-color 0.3s ease, color 0.3s ease;

}

/\* Dark theme for the body (can be toggled if combined with a theme feature) \*/

.dark-theme, body.dark-theme { /\* Added .dark-theme for consistency with UserDashboard if combined \*/

  background-color: #1a202c; /\* Dark background \*/

  color: #e2e8f0; /\* Light text \*/

}

/\* Styling for the main user list container \*/

.user-list-container {

  border-radius: 1rem; /\* Rounded corners \*/

  box-shadow: 0 4px 12px rgba(0, 0, 0, 0.1); /\* Soft shadow \*/

  transition: background-color 0.3s ease, color 0.3s ease;

  padding: 2rem;

}

/\* Specific styling for elements inside the container for light/dark mode \*/

.user-list-container.bg-white { /\* For light theme \*/

  background-color: #ffffff;

  color: #333333;

}

.user-list-container.dark\:bg-gray-800 { /\* For dark theme (tailwind class) \*/

  background-color: #2d3748;

  color: #f8f8f8;

}

/\* Input field basic styling \*/

input[type="text"] {

  border: 1px solid #ccc;

  padding: 0.75rem;

  border-radius: 0.5rem;

  font-size: 1rem;

  width: 100%; /\* Full width \*/

}

/\* Focus state for input field \*/

input[type="text"]:focus {

  outline: none;

  border-color: #3b82f6; /\* Blue border on focus \*/

  box-shadow: 0 0 0 3px rgba(59, 130, 246, 0.3); /\* Blue glow on focus \*/

}

/\* Specific styling for search input in dark mode \*/

.user-list-container .dark\:bg-gray-700 {

  background-color: #4a5568;

}

.user-list-container .dark\:text-gray-100 {

  color: #f7fafc;

}

/\* User list item styling \*/

.user-list ul li {

  /\* No specific styles needed beyond Tailwind classes applied in JSX \*/

}

/\* Ensuring font Inter is used for all elements \*/

body \* {

  font-family: "Inter", sans-serif;

}

README

React User Dashboard (useEffect Assignment)

This project demonstrates the usage of the useEffect hook in a React functional component for managing side effects, specifically data fetching and setting up/cleaning up timers. It simulates a user dashboard that displays a list of active users fetched from a mock API.

Assignment Tasks & Implementation Details

How useEffect is Used for Data Fetching

The useEffect hook is crucial for handling asynchronous operations like data fetching.

Initial Data Fetch:

A useEffect hook is used in the UserList component with an empty dependency array ([]). This ensures that the data fetching logic runs only once after the initial render of the component, mimicking componentDidMount behavior in class components.

Inside this useEffect, an asynchronous function getUsers is defined and immediately called.

getUsers calls a fetchUsersMockApi function, which simulates a network request using a Promise and setTimeout to introduce a 2-second delay.

States loading and error are used to manage the UI feedback during the fetch operation. setLoading(true) is called before the fetch, and setLoading(false) (along with setUsers or setError) is called in the finally block, ensuring the loading indicator is hidden regardless of success or failure.

The fetched data is stored in the users state using setUsers(data).

Handling Side Effects (Timers) with useEffect and Cleanup

useEffect is also ideal for managing non-data-fetching side effects like timers and event listeners, especially for ensuring proper cleanup.

Document Title Timer:

A separate useEffect hook is used to manage a timer that updates the document's title.

This effect sets up a setInterval that runs every 5 seconds. Inside the interval, document.title is updated to show the current number of fetched users.

The dependency array for this useEffect is [users.length]. This means the effect will re-run if the number of users changes, re-establishing the timer if necessary (though in this specific case, as data is fetched once, it primarily ensures the initial title update is based on the fetched data).

Cleanup Function:

Crucially, useEffect allows you to return a cleanup function. This function runs when the component unmounts (or before the effect re-runs if its dependencies change).

For the timer, the cleanup function calls clearInterval(timerId), where timerId is the ID returned by setInterval. This prevents memory leaks by stopping the timer when the UserList component is no longer rendered.

As part of the cleanup, document.title is also reset to 'React App' when the component unmounts, restoring the original title.

Search Functionality

An input field is used to allow users to type a search query.

The searchTerm state, managed by useState(''), stores the current value of the input field.

An onChange handler updates the searchTerm state as the user types.

The users array is filtered based on the searchTerm (case-insensitive) to create filteredUsers.

The UI then renders only the filteredUsers, providing dynamic search results.

Challenges Faced

Separating Concerns: Ensuring that data fetching and the timer were handled in separate useEffect calls (or at least with distinct purposes) helped in maintaining clear responsibilities for each side effect.

Correct Dependency Arrays: Carefully considering the dependency arrays for each useEffect was important. An empty array [] was used for the initial data fetch to run it only once, while [users.length] was used for the timer to ensure it reacts if the user count changes.

Proper Cleanup: Remembering to return a cleanup function from the useEffect for the timer was vital to prevent resource leaks and ensure the application behaves correctly when the component unmounts. Forgetting clearInterval is a common pitfall that can lead to unexpected behavior and performance issues.

This assignment provided valuable experience in leveraging useEffect for managing various side effects in React, emphasizing the importance of cleanup for robust application development.

Screenshot

