React Intermediate: State Management

React, React Context, Reducers, Zustand

***** App.tsx *****

```
import CounterNorm from "./state-management/counter/CounterNorm";
import EmotionStatus from "./state-management/emotions/EmotionStatus";
import LoginUseReducer from "./state-management/auth/LoginUseReducer";
import NavBar from "./state-management/tasks/NavBar";
import HomePage from "./state-management/tasks/HomePage";
import TasksProvider from "./state-management/tasks/TaskProvider";
import LoginStatus from "./state-management/auth/LoginStatus";
import TaskList from "./state-management/tasks/TaskList";
import AuthProvider from "./state-management/auth/AuthProvider";
import CounterWithReducer from "./state-management/counter/CounterWithReducer";
import CounterZustand from "./state-management/counter/CounterZustand";
import NavBarZustand from "./state-management/tasks/NavBarZustand";
import UserZustand from "./state-management/auth/UserZustand";
import EmotionStatusZustand from "./state-management/emotions/EmotionsZustand";
import TaskListZustand from "./state-management/tasks/TaskListZustand";
import useUserStore from "./state-management/auth/store";

function App() {
 const { user } = useUserStore();
 return (
   <div className="big_container">
     <div className="small_container">
       <h3>Normal Counter</h3>
       <CounterNorm />
     </div>

     <div className="small_container">
       <h3>Counter with Reducer</h3>
       <CounterWithReducer />
     </div>

     <div className="small_container">
       <h3>Task List</h3>
       <TaskList />
     </div>

     <div className="small_container">
       <h3>Login Status</h3>
```

```
          <LoginStatus />
        </div>

        <div className="small_container">
          <h3>Emotion Status</h3>
          <EmotionStatus />
        </div>

        <div className="small_container">
          <h3>Login (Youtube)</h3>
          <LoginUseReducer />
        </div>

        <div className="small_container">
          <h3>Tasks & Login: React Context</h3>
          <AuthProvider>
            <TasksProvider>
              <NavBar />
              <HomePage />
            </TasksProvider>
          </AuthProvider>
        </div>

        <hr />
        <h2>ZUSTAND</h2>
        <div className="small_container">
          <h3>Zustand Counter</h3>
          <h4>(Counter state shared with navbar counter.)</h4>
          <CounterZustand />
        </div>

        <div className="small_container">
          <h3>Zustand User Authentication</h3>
          <UserZustand />
          <h4>User: {user}</h4>
        </div>

        <div className="small_container">
          <h3>Zustand Emotion Status</h3>
          <EmotionStatusZustand />
        </div>

        <div className="small_container">
          <h3>Zustand Task Lists with Input</h3>
          <NavBarZustand />
          <TaskListZustand />
```

```
      </div>
    </div>
  );
}

export default App;
```

***** CounterNorm.tsx *****

```
// Reducer: a function that allows for centralization of state updates

import React, { useState } from "react";

const CounterNorm = () => {
  const [value, setValue] = useState(0);

  return (
    <div>
      Counter ({value})
      <button
        onClick={() => setValue(value + 1)}
        className="btn btn-primary mx-1"
      >
        Increment
      </button>
      <button onClick={() => setValue(0)} className="btn btn-primary mx-1">
        Reset
      </button>
    </div>
  );
};

export default CounterNorm;
```

***** CounterWithReducer.tsx *****

```
import React, { useReducer, useState } from "react";
import counterReducer from "./counterReducer";

const CounterWithReducer = () => {
  /* useReducer is a hook that allows for centralization of state updates
      it takes a reducer function and an initial state and returns an array:
     1. the current state
     2. a dispatch function that updates the state, dispatching an action */
```

```
 const [value, dispatch] = useReducer(counterReducer, 0);

 return (
   <div>
     Counter ({value})
     <button
       // dispatch an action to the reducer
       onClick={() => dispatch({ type: "INCREMENT" })}
       className="btn btn-primary mx-1"
     >
       Increment
     </button>
     <button
       onClick={() => dispatch({ type: "RESET" })}
       className="btn btn-primary mx-1"
     >
       Reset
     </button>
   </div>
 );
};

export default CounterWithReducer;
```

***** counterReducer.ts *****

```
interface Action {
   // ensures actions are only of these types, eliminating throwing errors
   type: 'INCREMENT' | 'RESET';
}

const counterReducer = (state: number, action: Action): number => {
   // INCREMENT is an arbitrary string that we use to identify the action
   if (action.type === 'INCREMENT') return state + 1;
   if (action.type === 'RESET') return 0;
   return state;
}

export default counterReducer;
```

***** store.tsx (for CounterZustand.tsx) *****

```
import { create } from "zustand";
import { mountStoreDevtool } from "simple-zustand-devtools";

interface CounterStore {
   counter: number;
   increment: () => void;
   reset: () => void;
}

/* passing the shape of the store, which is CounterStore
   arrow function takes set, which is a funciton for updating the state of the store
   it returns an object, so the {} is wrapped in () to avoid looking like a block
   of code

   - increment takes the current state and returns the new state.
   - set merges the property into the next state, so we do not have to ...spread things

   create returns a custom hook that we can use to access the store
*/

const useCounterStore = create<CounterStore>((set) => ({
   counter: 0,
   increment: () => set(state => ({ counter: state.counter + 1 })),
   reset: () => set(() => ({ counter: 0 })),
}));

/* if we are in development mode, we can mount the store devtool
this will allow us to see the state of the store in the browser
and also allow us to time travel debug with inspect - components */
if (process.env.NODE_ENV === "development") {
   mountStoreDevtool("Counter Store", useCounterStore);
}

export default useCounterStore;
```

***** CounterZustand.tsx *****

```
import useCounterStore from "./store";

const CounterZustand = () => {
 // this can be used in any component to get the state of the counter
 const { counter, increment, reset } = useCounterStore();

 return (
   <div>
     Counter ({counter})
```

```
      <button onClick={() => increment()} className="btn btn-primary mx-1">
        Increment
      </button>
      <button onClick={() => reset()} className="btn btn-primary mx-1">
        Reset
      </button>
    </div>
  );
};

export default CounterZustand;
```

***** TaskList.tsx *****

```
import { useContext, useReducer, useState } from "react";
import TasksContext from "./tasksContext";

export interface Task {
  id: number;
  title: string;
}

interface AddTask {
  type: "ADD";
  task: Task;
}

interface DeleteTask {
  type: "DELETE";
  taskId: number;
}

export type TaskAction = AddTask | DeleteTask;

// input state is a task array, and returns a taks array
const tasksReducer = (tasks: Task[], action: TaskAction): Task[] => {
  switch (action.type) {
    case "ADD":
      return [action.task, ...tasks];

    case "DELETE":
      return tasks.filter((task) => task.id !== action.taskId);
  }
};
```

```
const useTasks = () => useContext(TasksContext);

const TaskList = () => {
 const [tasks, dispatch] = useReducer(tasksReducer, []);

 return (
   <>
     <button
       onClick={() =>
         dispatch({
           type: "ADD",
           task: { id: Date.now(), title: "Task " + Date.now() },
         })
       }
       className="btn btn-primary my-3"
     >
       Add Task
     </button>
     <ul className="list-group">
       {tasks.map((task) => (
         <li
           key={task.id}
           className="list-group-item d-flex justify-content-between align-items-center"
         >
           <span className="flex-grow-1">{task.title}</span>
           <button
             className="btn btn-outline-danger"
             onClick={() => dispatch({ type: "DELETE", taskId: task.id })}
           >
             Delete
           </button>
         </li>
       ))}
     </ul>
   </>
 );
};

export default TaskList;

/* This is a React functional component named TaskList which uses the useReducer
and useState hooks from the React library for state management. The tasks are
managed by a reducer function called tasksReducer that's imported from another
module. This component is intended to render a list of tasks and provide the
functionality to add and delete tasks.
```

```
TaskList is a functional component. Inside this component, the useReducer
hook is used to create the tasks state variable and the dispatch function.
The initial state of tasks is an empty array, and tasksReducer is the function
that will manage updates to the tasks state.

The "Add Task" button dispatches an action of type "ADD" to the reducer when
clicked. The dispatch function is used to send this action object to the
tasksReducer. The action object includes a task object with an id (current
timestamp) and title ("Task " followed by the current timestamp).

The tasks are rendered in an unordered list (ul). For each task in the tasks
array, a list item (li) is created with a key of task.id. Each list item contains
the task title and a "Delete" button. The button dispatches a "DELETE" action with
the task's ID when clicked.
*/
```

***** TaskListContext.tsx *****

```
import { useContext } from "react";
import LoginContext from "../auth/loginContext";
import TasksContext from "./tasksContext";

// Do no tneed anymore now that we have useTasks hook and TasksProvider
// const useTasks = () => useContext(TasksContext);

const useTasks = () => useContext(TasksContext);

const TaskListContext = () => {
 const { tasks, dispatch } = useTasks();
 // Tried to use useAuth, but it didn't work.
 const { user } = useContext(LoginContext);

 return (
   <>
     <p>User: {user}</p>
     <button
       onClick={() =>
         dispatch({
           type: "ADD",
           task: { id: Date.now(), title: "Task " + Date.now() },
         })
       }
       className="btn btn-primary my-3"
```

```
      >
        Add Task
      </button>
      <ul className="list-group">
        {tasks.map((task) => (
          <li
            key={task.id}
            className="list-group-item d-flex justify-content-between align-items-center"
          >
            <span className="flex-grow-1">{task.title}</span>
            <button
              className="btn btn-outline-danger"
              onClick={() => dispatch({ type: "DELETE", taskId: task.id })}
            >
              Delete
            </button>
          </li>
        ))}
      </ul>
    </>
  );
};

export default TaskListContext;
```

***** tasksContext.ts *****

```
import React, { Dispatch } from "react";
import { Task, TaskAction } from "./TaskProvider";

interface TasksContextType {
   tasks: Task[];
   dispatch: Dispatch<TaskAction>;
}

const TasksContext = React.createContext<TasksContextType>({} as TasksContextType)

export default TasksContext;
```

***** TaskProvider.tsx *****

```
import { ReactNode, useReducer } from "react";
import TasksContext from "./tasksContext";

export interface Task {
 id: number;
 title: string;
}

interface AddTask {
 type: "ADD";
 task: Task;
}

interface DeleteTask {
 type: "DELETE";
 taskId: number;
}

export type TaskAction = AddTask | DeleteTask;

// input state is a task array, and returns a taks array
const tasksReducer = (tasks: Task[], action: TaskAction): Task[] => {
 switch (action.type) {
   case "ADD":
     return [action.task, ...tasks];

   case "DELETE":
     return tasks.filter((task) => task.id !== action.taskId);
 }
};

interface Props {
 // always be sure to import the correct ReactNode, from React
 children: ReactNode;
}

const TasksProvider = ({ children }: Props) => {
 const [tasks, dispatch] = useReducer(tasksReducer, []);
 return (
   <TasksContext.Provider value={{ tasks, dispatch }}>
     {children}
   </TasksContext.Provider>
 );
};

export default TasksProvider;
```

***** NavBar.tsx *****

```
import { useContext } from "react";
import TasksContext from "./tasksContext";
import LoginStatusContext from "../auth/LoginStatusContext";

const NavBar = () => {
 const { tasks } = useContext(TasksContext);

 return (
   <nav className="navbar d-flex justify-content-between">
     <span className="badge text-bg-secondary">
       Tasks Total: {tasks.length}
     </span>
     <LoginStatusContext />
   </nav>
 );
};

export default NavBar;
```

***** HomePage.tsx *****

```
import TaskListContext from "./TaskListContext";

const HomePage = () => {
 return <TaskListContext />;
};

export default HomePage;
```

***** store.ts (for TaskListZstand.tsx) *****

```
import { create } from "zustand";

/*
export interface Task {...} and interface TasksStore {...}
These lines define TypeScript interfaces to provide type checking. The Task
interface describes what a Task object should look like, while TasksStore
describes the structure and types of the Zustand store.
*/

export interface Task {
 id: number;
 title: string;
}
```

```
interface TasksStore {
 tasks: Task[];
 taskTitle: string;
 addTask: () => void;
 deleteTask: (id: number) => void;
 setTaskTitle: (title: string) => void;
}

/*
const useTasksStore = create<TasksStore>((set) => {...});
Here, the create function is called to create a new Zustand store. The TasksStore
interface is passed as a type parameter to ensure the created store conforms to
the defined structure. The create function takes a callback function as an argument,
which receives a set function to be used for updating the store's state.

tasks: [], taskTitle: "",
Inside the callback, initial states for the tasks array and taskTitle string are set
to an empty array and an empty string, respectively.

addTask: () => set((state) => {...}),
This line defines an addTask action for adding a new task to the tasks array. The
set function updates the state by taking a function that receives the current state
and returns the new state. A new task is created with a unique ID (using Date.now())
and the current taskTitle. The new task is added to the end of the tasks array and
taskTitle is reset to an empty string.

deleteTask: (id) => set((state) => {...}),
This line defines a deleteTask action for deleting a task from the tasks array. The
set function updates the tasks array to a new array that does not include the task
with the given ID.

setTaskTitle: (title) => set(() => ({ taskTitle: title }))
This line defines a setTaskTitle action for updating the taskTitle state. The set
function updates the taskTitle state to the passed title.
*/

const useTasksStore = create<TasksStore>((set) => ({
 tasks: [],
 taskTitle: "",
 addTask: () => set((state) => {
   const newTask = { id: Date.now(), title: state.taskTitle };
   return { tasks: [...state.tasks, newTask], taskTitle: "" };
 }),
 deleteTask: (id) => set((state) => ({ tasks: state.tasks.filter((task) => task.id !== id) })),
 setTaskTitle: (title) => set(() => ({ taskTitle: title }))
}));

export default useTasksStore;
```

***** TaskListZustand.tsx *****

```
import useTasksStore from "./store";
import useUserStore from "../auth/store";

const TaskListZustand = () => {
 /*
   const { tasks, addTask, deleteTask, taskTitle, setTaskTitle } = useTasksStore();
   const { user, login, logout } = useUserStore();
   Here, we're using destructuring assignment to get various properties from our
   Zustand stores. From useTasksStore, we're getting the current list of tasks,
   the current task title, and the functions to add a task, delete a task, and set
   the task title. From useUserStore, we're getting the current user and the functions
   to log in and log out.
   */

 const { tasks, addTask, deleteTask, taskTitle, setTaskTitle } =
   useTasksStore();
 const { user, login, logout } = useUserStore();

 return (
   /* This is the render function of the TaskListZustand component. It specifies what the
component should output to the DOM. */
   <>
     {/* This line renders a paragraph with the current user's name. */}
     <p>User: {user}</p>

     {/* This line renders an input element. The value of the input is bound to the
     taskTitle from the Zustand store, and whenever the value changes due to
     user input, it calls the setTaskTitle function to update the task title in
     the store. */}
     <input
       type="text"
       value={taskTitle}
       onChange={(e) => setTaskTitle(e.target.value)}
     />

     {/* This line renders a button that, when clicked, calls the addTask function
     from the Zustand store. This function will create a new task with the
     current taskTitle and add it to the tasks list. The ul element contains a
     map function which maps each task in the tasks array to a li element,
     displaying the task title and a Delete button. */}
     <button onClick={addTask} className="btn btn-primary my-3">
       Add Task
     </button>
     <ul className="list-group">
       {tasks.map((task) => (
         <li
           key={task.id}
           className="list-group-item d-flex justify-content-between align-items-center"
         >
           <span className="flex-grow-1">{task.title}</span>

           {/* Inside each li, there is a Delete button that, when clicked, calls
           the deleteTask function from the Zustand store with the id of the
           task. This function will remove the task with the given id from the
           tasks list. */}
           <button
             className="btn btn-outline-danger"
             onClick={() => deleteTask(task.id)}
           >
             Delete
           </button>
         </li>
       ))}
     </ul>
   </>
 );
};

export default TaskListZustand;
```

***** NavBarZustand.tsx *****

```
import useCounterStore from "../counter/store";
import UserZustand from "../auth/UserZustand";
import useTasksStore from "./store";

const NavBarZustand = () => {
  const { tasks } = useTasksStore();
  // Using store to get counter state, not as object now, but as a value
  const counter = useCounterStore((s) => s.counter);

  return (
    <nav className="navbar d-flex justify-content-between">
      <span className="badge text-bg-secondary">
```

```
        Tasks Total: {tasks.length}
      </span>
      <span className="badge text-bg-secondary">Counter: {counter}</span>
      <UserZustand />
    </nav>
  );
};

export default NavBarZustand;
```

***** LoginStatus.tsx *****

```
import { useReducer } from "react";
import { loginReducer } from "./AuthProvider";

const LoginStatus = () => {
  const [user, dispatch] = useReducer(loginReducer, "");

  if (user)
    return (
      <>
        <div>
          <button
            onClick={() => dispatch({ type: "LOGOUT" })}
            className="btn btn-primary mx-1"
          >
            Logout
          </button>
          <span className="mx-2">{user}</span>
        </div>
      </>
    );
  return (
    <div>
      <button
        onClick={() => dispatch({ type: "LOGIN", username: "Evan.Marie" })}
        className="btn btn-primary mx-1"
      >
        Login
      </button>
    </div>
  );
};

export default LoginStatus;
```

***** LoginStatusContext.tsx *****

```
import { useContext } from "react";
import LoginContext from "./loginContext";

const LoginStatusContext = () => {
 // Tried to use AuthContext, but it didn't work.
 // const { user, dispatch } = useAuth();
 const { user, dispatch } = useContext(LoginContext);

 if (user)
   return (
     <>
       <div>
         <span className="mx-2">{user}</span>
         <button
           onClick={() => dispatch({ type: "LOGOUT" })}
           className="btn btn-primary mx-1"
         >
           Logout
         </button>
       </div>
     </>
   );
 return (
   <div>
     <button
       onClick={() => dispatch({ type: "LOGIN", username: "Evan.Marie" })}
       className="btn btn-primary mx-1"
     >
       Login
     </button>
   </div>
 );
};

export default LoginStatusContext;
```

***** loginContext.ts *****

```
import React, { Dispatch } from "react";
import { authAction } from "./AuthProvider";
```

```
interface LoginContextType {
   user: string;
   dispatch: Dispatch<authAction>;
}

const LoginContext = React.createContext<LoginContextType>({} as LoginContextType)

export default LoginContext;
```

***** AuthProvider.tsx *****

```
import { ReactNode, useReducer } from "react";
import LoginContext from "./loginContext";

interface LoginAction {
 type: "LOGIN";
 username: string;
}

interface LogoutAction {
 type: "LOGOUT";
}

export type authAction = LoginAction | LogoutAction;

export const loginReducer = (state: string, action: authAction): string => {
 if (action.type === "LOGIN") return action.username;
 if (action.type === "LOGOUT") return "";
 return state;
};

interface Props {
 // always be sure to import the correct ReactNode, from React
 children: ReactNode;
}

const AuthProvider = ({ children }: Props) => {
 const [user, dispatch] = useReducer(loginReducer, "");
 return (
   <LoginContext.Provider value={{ user, dispatch: dispatch }}>
     {children}
   </LoginContext.Provider>
 );
};
```

```
export default AuthProvider;
```

***** useAuth.ts *****

```
import { useContext } from "react";
import LoginContext from "./loginContext";

const useAuth = () => useContext(LoginContext);

export default useAuth;
```

***** store.ts (for UserZustand.tsx) *****

```
import { create } from "zustand";

interface UserStore {
   user: string;
   login: (username: string) => void;
   logout: () => void;
}

/* instead of state as the parameter for set, we can use () to indicate that we
do not need the current state, since we are just setting the user to the username
and we are not computing the next state based on the current state */

const useUserStore = create<UserStore>(set => ({
 user: '',
 login: username => set(() => ({ user: username })),
 logout: () => set(() => ({ user: '' }))
}));

export default useUserStore;
```

***** UserZustand.tsx *****

```
import useUserStore from "./store";

const UserZustand = () => {
 const { user, login, logout } = useUserStore();

 if (user)
   return (
     <>
       <div>
         <button onClick={() => logout()} className="btn btn-primary mx-1">
```

```
          Logout
        </button>
        {/* <span className="mx-2">{user}</span> */}
      </div>
    </>
  );
 return (
   <div>
     <button
       onClick={() => login("Evan.Marie")}
       className="btn btn-primary mx-1"
     >
       Login
     </button>
   </div>
 );
};

export default UserZustand;
```

***** useLogin.ts *****

```
/*
from Youtube: https://www.youtube.com/watch?v=9KzQ9xFSAEU

This function can be used to simulate a login operation in a testing
environment or while developing the UI of a login feature, before a real
server-side authentication system is implemented.

This defines a TypeScript type named Props, which is an object with two
properties: username and password. Both properties are of the string type.
This Props type is used to specify the type of the argument that the
useLogin function expects.
*/

type Props = {
   username: string;
   password: string;
}

/*
The useLogin function is an asynchronous function that takes an object as
an argument. This object must have a username and a password property, as
defined by the Props type. The function returns a Promise that doesn't
resolve with any value (hence void).
```

```
Inside the function, a setTimeout is used to simulate a delay that you
might experience when making a real asynchronous request to a server.
After 1 second (1000 milliseconds), the callback function passed to
setTimeout is executed.

In this callback function, if the username is "evan" and the password
is "password", the Promise is resolved using the resolve function. If
the username and password don't match these values, the Promise is
rejected using the reject function.
*/


async function useLogin({ username, password }: Props) {
 return new Promise<void>((resolve, reject) => {
   setTimeout(() => {
     if (username === 'evan' && password === 'password') {
       resolve();
     } else {
       reject();
     }
   }, 1000);
 });
}

export default useLogin;
```

***** LoginUseReducer.tsx *****

```
/*
from Youtube: https://www.youtube.com/watch?v=9KzQ9xFSAEU
*/

import React, { useReducer } from "react";
import useLogin from "./useLogin";

/*
The LoginUseReducer function represents a component that provides a
form for user login, handles login and logout actions, and displays a
welcome message to the logged-in user.

Here an interface LoginState is defined to specify the shape of the state
object. The initialState is the initial state for the login form, which
is an object of type LoginState.
*/
```

```
const initialState: LoginState = {
 username: "",
 password: "",
 isLoading: false,
 error: "",
 isLoggedIn: false,
 variant: "login",
};

interface LoginState {
 username: string;
 password: string;
 isLoading: boolean;
 error: string;
 isLoggedIn: boolean;
 variant: "login" | "forgetPassword";
}

/*
A LoginAction type is defined for actions that can be dispatched to the reducer.
It can either be an object with a type of "login", "success", "error", or "logOut",
or an object with a type of "field" and additional properties fieldName and payload.
*/

type LoginAction =
 | { type: "login" | "success" | "error" | "logOut" }
 | { type: "field"; fieldName: string; payload: string };

/*
Defining the Reducer Function:
The loginReducer function is defined to handle state changes based on dispatched
actions. It takes the current state and an action, and returns a new state. Each
case in the switch statement corresponds to a different action type.
*/

function loginReducer(state: LoginState, action: LoginAction) {
 switch (action.type) {
   case "field": {
     return {
       ...state,
       [action.fieldName]: action.payload,
     };
   }
   case "login": {
     return {
       ...state,
       error: "",
       isLoading: true,
     };
   }
   case "success": {
     return {
       ...state,
       isLoggedIn: true,
       isLoading: false,
     };
   }
   case "error": {
     return {
       ...state,
       error: "Incorrect username or password!",
       isLoggedIn: false,
       isLoading: false,
       username: "",
       password: "",
     };
   }
   case "logOut": {
     return {
       ...state,
       isLoggedIn: false,
     };
   }
   default:
     return state;
 }
}

/*
Creating the LoginUseReducer Component:
In the LoginUseReducer component, the useReducer hook is used to create the state
variable and the dispatch function. The loginReducer is the function that manages
updates to the state, and initialState is the initial value of the state.

The onSubmit function is an asynchronous function that handles the form submission.
It first dispatches a "login" action, then it calls the useLogin function with the
username and password from the state. If useLogin resolves, a "success" action is
dispatched. If useLogin rejects, an "error" action is dispatched.
*/

export default function LoginUseReducer() {
 const [state, dispatch] = useReducer(loginReducer, initialState);
```

```
 const { username, password, isLoading, error, isLoggedIn } = state;

 const onSubmit = async (e: React.FormEvent) => {
   e.preventDefault();

   dispatch({ type: "login" });

   try {
     await useLogin({ username, password });
     dispatch({ type: "success" });
   } catch (error) {
     dispatch({ type: "error" });
   }
 };

 /*
Rendering the Component:
In the return statement, the component renders a form for login if the user is
not logged in (isLoggedIn is false), and a welcome message and a "Log Out"
button if the user is logged in (isLoggedIn is true).

In the login form, the username and password fields are controlled inputs that
dispatch "field" actions when their values change. The isLoading state variable
is used to disable the submit button and change its text while the login operation
is in progress.

If the error state variable is not an empty string, an error message is displayed
above the form.
*/
 return (
   <div className="App">
     <div className="login-container">
       {isLoggedIn ? (
         <>
           <h1>Welcome {username}!</h1>
           <button onClick={() => dispatch({ type: "logOut" })}>
             Log Out
           </button>
         </>
       ) : (
         <form className="form" onSubmit={onSubmit}>
           {error && <p className="error">{error}</p>}
           <p>Enter username and password:</p>
           <input
             type="text"
             placeholder="username"
```

```
            value={username}
            onChange={(e) =>
              dispatch({
                type: "field",
                fieldName: "username",
                payload: e.currentTarget.value,
              })
            }
          />
          <input
            type="password"
            placeholder="password"
            autoComplete="new-password"
            value={password}
            onChange={(e) =>
              dispatch({
                type: "field",
                fieldName: "password",
                payload: e.currentTarget.value,
              })
            }
          />
          <button
            type="submit"
            disabled={isLoading}
            className="btn btn-primary mx-1"
          >
            {isLoading ? "Logging in..." : "Login"}
          </button>
        </form>
      )}
    </div>
  </div>
 );
}

/*
More of "FIELD ACTION":
A "field" action in this context is an action object that is used to update the value of a field
in the state. In this case, the fields are username and password in the state object.

Here's the definition of a "field" action from the LoginAction type:
   { type: "field"; fieldName: string; payload: string }

   - type: The type of the action. In this case, it's a string with the value
     "field".
   - fieldName: The name of the field to update. It's a string that should match
     one of the keys in the state object (username or password).
   - payload: The new value for the field. It's a string that will be used to
     update the value of the field.

When a "field" action is dispatched to the reducer, the reducer updates the value
of the specified field in the state. Here's the relevant code from the loginReducer
function:
   case "field": {
       return {
           ...state,
           [action.fieldName]: action.payload,
       };
       }

This code creates a new object that is a copy of the current state, but with the
specified field updated to the new value.

Here's an example of a "field" action being dispatched when the value of the
username input field changes:
   onChange={(e) =>
       dispatch({
           type: "field",
           fieldName: "username",
           payload: e.currentTarget.value,
       })
       }

In this code, an onChange event handler dispatches a "field" action with the
fieldName set to "username" and the payload set to the current value of the input
field. This will update the username field in the state with the current value of
the input field.
*/
```

***** EmotionStatus.tsx *****

```
import React from "react";
import { useReducer } from "react";
import emotionReducer, { EmotionAction } from "./emotionReducer";

const EmotionStatus = () => {
 const [message, dispatch] = useReducer(emotionReducer, "");

 const handleButtonClick = (type: EmotionAction["type"]) => {
   dispatch({ type });
 };

 return (
   <>
     <div className="emotion_buttons">
       <button
         onClick={() => handleButtonClick("HAPPY")}
         className="btn btn-primary mx-1"
       >
         😃 HAPPY!
       </button>
       <button
         onClick={() => handleButtonClick("SAD")}
         className="btn btn-primary mx-1"
       >
         🤕 sad...
       </button>
       <button
         onClick={() => handleButtonClick("FEISTY")}
         className="btn btn-primary mx-1"
       >
         😜 FEISTY!
       </button>
       <button
         onClick={() => handleButtonClick("MEH")}
         className="btn btn-primary mx-1"
       >
         😒 meh...
       </button>
     </div>
     <div className="emotion_message">
       <span className="mx-2">{message}</span>
     </div>
   </>
 );
};

export default EmotionStatus;

/* - The EmotionStatus functional component is defined. It represents the
    component responsible for displaying buttons and the corresponding
    message based on the selected emotion.

   - Inside the component, the useReducer hook is used to initialize the
   state and dispatch function. It takes two arguments: the emotionReducer
   function (imported from the reducer file) and the initial state value,
   which is an empty string "". The hook returns the current state value
   (message) and the dispatch function.

   - The handleButtonClick function is defined to handle button clicks. It
   takes a parameter type with the type of EmotionAction["type"]. When a
   button is clicked, this function is called with the corresponding action
   type, and it dispatches the action by invoking dispatch with the action
   object { type }.

   - The component's JSX code renders a series of buttons representing
   different emotions. Each button has an onClick event handler that invokes
   the handleButtonClick function with the corresponding action type ("HAPPY",
   "SAD", "FEISTY", or "MEH").

   - The component also includes a <span> element to display the current
   message state value. This element is rendered within a <div> element with
   the class name "emotion_message".

   - The component uses the useReducer hook to manage state and dispatch
   actions based on button clicks. The handleButtonClick function dispatches
   the corresponding action type, and the reducer (defined in the reducer
   file) handles the actions to update the state accordingly. The
   component renders buttons and displays the message based on the
   current state value.

*/
```

***** emotionsReducer.tsx *****

```
export type EmotionAction = {
 type: "HAPPY" | "SAD" | "FEISTY" | "MEH";
};

const emotionReducer = (state = "", action: EmotionAction) => {
   switch (action.type) {
       case "HAPPY":
           return "I am so happy today!"

       case "SAD":
           return "I am so sad today..."

       case "FEISTY":
           return "I am feeling feisty today!"

       case "MEH":
            return "I am just meh..."

        default:
            return state;
    }
}

export default emotionReducer;

/*
- The EmotionAction type is defined as a TypeScript type. It specifies
that an action should have a type property with one of the values: "HAPPY",
"SAD", "FEISTY", or "MEH". This type is used to enforce type safety and
ensure that only valid action types are used.

- The emotionReducer function is defined, which takes two parameters: state
and action. The state parameter represents the current state, and the action
parameter represents the action being dispatched.

- Inside the emotionReducer function, a switch statement is used to handle
different action types. The action.type property is evaluated to determine
which case matches the type of the action being dispatched.

- In each case, the reducer returns a new state based on the action type.
For example, if the action type is "HAPPY", the reducer returns the string
"I am so happy today!". Similarly, for other action types like "SAD", "FEISTY",
and "MEH", appropriate strings are returned.

- If the action type does not match any of the cases, the default case is executed,
and the current state is returned as is.

- The reducer is responsible for handling actions and updating the state
accordingly. It receives the current state and an action, and based on the
action type, it returns a new state. The use of the switch statement allows
the reducer to easily handle different action types and define the
corresponding behavior for each type.

- It's important to note that the reducer function should be a pure function,
meaning it should not modify the original state. Instead, it should return a
new state object or value based on the provided inputs.
*/
```

***** store.ts (for EmotionsZustand.tsx) *****

```
import { create } from "zustand";
```

```
/*
The EmotionsStore interface is declared to define the shape of the state store.
It includes two properties: emotion and message, and two methods for updating these
properties: setEmotion and setMessage.
*/

export interface EmotionsStore {
 emotion: "HAPPY" | "SAD" | "FEISTY" | "MEH" | null;
 message: string;
 setEmotion: (emotion: EmotionsStore["emotion"]) => void;
 setMessage: (message: string) => void;
}

/*
const useEmotionsStore = create<EmotionsStore>((set) => ({... creates a new Zustand
store using the create function. The set function is used to update the state in the
store.

   - emotion: null, and message: "", are the initial states for the emotion and
       message properties.

   - setEmotion: (emotion: EmotionsStore["emotion"]) => set({ emotion }) is a
       function that takes an emotion as argument and uses the set function to
       update the emotion property in the state.

   - Similarly, setMessage: (message: string) => set({ message }) is a function that
       takes a message string and uses the set function to update the message property
       in the state.
*/

const useEmotionsStore = create<EmotionsStore>((set) => ({
 emotion: null,
 message:"",
 setEmotion: (emotion: EmotionsStore["emotion"]) => set({ emotion }),
 setMessage: (message: string) => set({ message }),
}));

/*
getEmotionMessage function is a utility function that takes an emotion and returns a
string message associated with the given emotion.
*/

export const getEmotionMessage = (emotion: EmotionsStore["emotion"]) => {
   switch (emotion) {
       case "HAPPY":
           return "I am so happy today!"

       case "SAD":
           return "I am so sad today..."

       case "FEISTY":
           return "I am feeling feisty today!"

       case "MEH":
           return "I am just meh..."

       default:
           return "";
   }
};

export default useEmotionsStore;
```

***** EmotionsZustand.tsx *****

```
import useEmotionsStore, { EmotionsStore, getEmotionMessage } from "./store";

/*
const { emotion, setEmotion, setMessage } = useEmotionsStore(); This line is using
the useEmotionsStore hook to access the current state of the Zustand store and the
functions for updating the state.a
*/

const EmotionStatusZustand = () => {
 const { emotion, setEmotion, setMessage } = useEmotionsStore();

 /*
handleButtonClick is a function that takes an emotion as argument.
Inside this function, it calls setEmotion and setMessage, effectively
updating the emotion and message properties in the state store.
*/

 const handleButtonClick = (emotion: EmotionsStore["emotion"]) => {
   setEmotion(emotion);
   setMessage(emotion ? getEmotionMessage(emotion) : "");
 };

 /*
The return statement is the render method of this functional component. It includes
four buttons, each associated with a different emotion. When a button is clicked, the
handleButtonClick function is called with the corresponding emotion.
In the div with the class emotion_message, it displays the message associated with
the current emotion in the state. It uses a ternary operator to check if emotion
exists, and if it does, it calls getEmotionMessage to get the associated message.
If emotion does not exist (null), it displays an empty string.
*/

 return (
   <>
     <div className="emotion_buttons">
       <button
         onClick={() => handleButtonClick("HAPPY")}
         className="btn btn-primary mx-1"
       >
         😃 HAPPY!
       </button>
       <button
         onClick={() => handleButtonClick("SAD")}
         className="btn btn-primary mx-1"
       >
         🤕 sad...
       </button>
       <button
         onClick={() => handleButtonClick("FEISTY")}
         className="btn btn-primary mx-1"
       >
         😜 FEISTY!
       </button>
       <button
         onClick={() => handleButtonClick("MEH")}
         className="btn btn-primary mx-1"
       >
         😒 meh...
       </button>
     </div>
     <div className="emotion_message">
       <span className="mx-2">
         {emotion ? getEmotionMessage(emotion) : ""}
       </span>
     </div>
   </>
 );
};

export default EmotionStatusZustand;
```

***** About React Context *****

```
/*
React Context is a feature in React that allows you to pass data through the
component tree without having to pass it down manually through props at every
level. It's particularly useful when you need to share global data or state,
like a user's authentication status, theme settings, or a global store. Here
are the main aspects of React Context:

Creating a Context:
To create a context, you use the React.createContext function. It returns
a Context object with two main components: Provider and Consumer (or
useContext hook).
*/

const MyContext = React.createContext();

/*
Context Provider:
The Provider is a component that wraps the part of your component tree that
needs access to the context data. It accepts a prop called value, which is the
data you want to pass to the components in the tree.
*/

<MyContext.Provider value={someData}>
  {/* Your component tree that needs access to the context data */}
</MyContext.Provider>;

/*
Accessing Context Data:
There are two main ways to access the data provided by a context:

Context Consumer: This is a component that can be used to access the context
value directly within the render method. It accepts a function as a child,
and that function receives the context value as an argument.
*/

<MyContext.Consumer>
  {(contextValue) => {
    // You can use the contextValue in your JSX here
  }}
</MyContext.Consumer>;

/*
useContext Hook: In functional components, you can use the useContext hook to
access the context value. This hook accepts the context object as an argument
```

```
and returns the current context value.
*/

import { useContext } from "react";

function MyComponent() {
  const contextValue = useContext(MyContext);
  // You can use the contextValue in your JSX or logic here
}

/*
Updating Context Data:
To update the context data, you can either change the value passed to the
Provider or, more commonly, use a combination of context and state management
techniques, like using the context to provide a state and a dispatch function
(similar to how the useReducer hook works).

****************************************************************************************
WHEN TO and WHEN NOT TO USE REACT CONTEXT:

* CLIENT STATE: the data that represents the state of the client / UI, i.e. current
  user, selected theme, etc.
  - For managing client state, local state can be saved in a component with useState or
    useReducer (for more complicated state management), and React Context can be used
    to share it with the child component.
  - This often involves lifting the state up to a parent component so it can be
    shared with its children

* SERVER STATE: the data fetched from the backend
  - not a good place to use React Context, which would complicate the component
    tree very quickly.
  - React Query is a better choice here.

- Contexts should have a single responsibility and be split up to minimize re-rendering
- State Management Libraries: Redux, MobX, Recoil, xState, Zustand, etc. (Zustand is
  simplest and works for most applications.)

****************************************************************************************
CHATGPT EXPLAINS WHEN TO AND NOT TO USE REACT CONTEXT:

React Context is a built-in state management feature in React that helps pass data down
the component tree without having to pass props manually at every level. The context
system can be a great tool to manage state, but it's not always the right tool for every
job.

* Here are some scenarios where you may want to use React Context:
```

- Prop Drilling: If you're experiencing a problem with "prop drilling", where props need to be passed through multiple components before reaching the one that actually uses the prop, then React Context can be a good solution. It enables you to share value or state to the components that need it without going through intermediates. This keeps your code cleaner and easier to manage.

- Shared State: When you have global state that multiple components or component trees need access to. For example, if you have user authentication information that many components need to access, React Context is a good tool to use.

- Theme Management: If you are changing the look and feel of your application and the changes need to be reflected across multiple components, React Context is a great choice. You can store the current theme in the context and use it throughout the application.

* However, there are also scenarios where React Context may not be the best choice:

- High-frequency Updates: If your state changes very frequently (like the position of a cursor in a drawing app), using context can cause unnecessary re-renders and negatively impact performance. In this case, you might want to explore other alternatives like storing the state locally or using other state management libraries optimized for frequent updates.

- Very Large Applications: In large-scale applications with complex state management needs, Redux or MobX might be better suited because they provide more robust solutions for managing state with middlewares, and they have better tools for debugging state changes.

- Small, isolated state: If a piece of state only affects one or a small number of components, using local component state with useState or useReducer would be a more straightforward choice. Using context in this case would be an overkill and may make the component harder to understand and reuse.

- Very Local or Temporary States: Things like form input values, hover states, or toggled visibility are typically best kept in local component state. They're not usually beneficial to other parts of the app, so there's no need to use context.

* Remember that Context is just a tool, and like any tool, its efficacy depends on the situation. It can sometimes be the perfect solution, but other times there may be better options available. When you're considering whether to use Context, it can help to think about what problem you're trying to solve and whether Context is the best tool for that specific problem.

****

CHAT GPT EXPLAINS REDUX:

* Redux is a predictable state container for JavaScript applications. It was designed to help you manage global state in an application, particularly when dealing with complex flows of data and intricate UIs. Redux was inspired by Facebook's Flux architecture and influenced by functional programming concepts, especially the Elm architecture.

** FLUX is an application architecture for building client-side web applications. It was developed by Facebook to complement React's components by utilizing a unidirectional data flow. It's more of a pattern rather than a formal framework, and you can use it immediately with React in your applications.

Flux is comprised of a few parts: Actions, Dispatcher, Stores, and Views (React components). The flow of data in Flux is as follows:

* Actions: User interactions in the view cause dispatches of actions (simple objects containing the new data and type of action).
* Dispatcher: Acts as a central hub where callbacks are registered. Each store registers itself and provides a callback.
* Stores: They contain the application's state and logic. Whenever an action is dispatched, the store's callback is invoked, and depending on the action type, it will execute some logic and update the state.
* Views: React components grab the state from the Stores and re-render. They also pass down callback functions to the child components to propagate new actions.

The dispatcher, stores, and views are independent nodes with distinct inputs and outputs, and actions flowing in a single direction, which keeps the system easier to reason about.

** ELM is a functional language that compiles to JavaScript, and it's known for its strong type safety and friendly error messages. It was designed to build reliable, robust, and efficient web applications.

Elm enforces a simple, yet strict architecture pattern called The Elm Architecture (TEA), comprising three fundamental parts:

* Model: The state of your application.
* Update: A function to update your state with some new data. This function is pure, meaning given the same input, it will always return the same output without producing any side effects.
* View: A function to render HTML based on the state.

These three parts are wired together in a cyclical pattern: user interaction in the View generates a command for the Update, the Update function processes commands and updates the Model, and the changed Model triggers a re-render of the View.

Elm is commonly praised for its performance and simplicity, as well as for its innovative features like time-traveling debugger, where developers can go back and forth in their code to inspect their app at different points in time. It has influenced JavaScript frameworks like Redux and has been used as an alternative to JavaScript for building web frontends.

* Redux is most often used with libraries like React and Angular, but it can be used with any view library. It is tiny (about 2KB) and has no dependencies.

* Here's a brief overview of some key concepts in Redux:

  - Store: The Redux store is a JavaScript object that holds the global state of the application. It is the single source of truth for state within your application.

  - Actions: Actions are plain JavaScript objects that represent what happened in the app. They are the only way you can send data (payload) to the Redux store. Every action must have a type field which tells what kind of action it is.

  - Reducers: Reducers are pure functions that take the current state of the application and an action, then return a new state. They describe how the application's state changes in response to actions sent to the store.

* The primary use case for Redux is managing complex state interactions that are hard to express with React’s component state. It is also handy when you are dealing with shared state that needs to be accessed by multiple components.

* Here's when you might want to use Redux:

  - Complex state interactions: If you have actions that have side effects or are asynchronous (like network requests), or if multiple places need to respond to the same action, Redux can be a good choice.

  - Shared, global state: If you have state that needs to be shared amongst many components, or different parts of the state tree that need to be related, Redux can provide a central store for all of this state.

  - Performance with many components: If you have a high number of components that need to be aware of state, Redux can help optimize performance by avoiding the need for prop drilling and unnecessary component re-renders.

  - Developer tooling and middleware: Redux has great developer tools that allow you to track when, where, why, and how your application’s state changed. Redux’s middleware also allows you to write async logic that interacts with the store.

* On the other hand, for smaller applications, or applications with a simple state,

```
  Redux might be overkill and could add unnecessary complexity to your app. For such
  applications, using local component state or React's Context API might be a better
  choice.

* As with any tool, it's important to consider the trade-offs and choose the best
  tool for your specific needs.


*/
```