Concept of Using Multiple Reducers in Redux

In Redux, reducers are pure functions that determine how the application's state changes in response to dispatched actions. When an application grows in complexity, having a single reducer to handle all state changes can become unmanageable. This is where the concept of using multiple reducers comes into play.

- Redux allows splitting the logic of reducers into multiple smaller reducers, each responsible for managing a specific part of the application's state.
- These smaller reducers are then combined into a single root reducer using combineReducers from Redux. Each reducer handles actions related to a specific feature or a piece of state.
- Using multiple reducers helps maintain cleaner, more scalable, and more modular code in Redux applications, making it easier to manage state as the complexity of the app increases.

Example

1. **Create a new directory and set up a new project**

   npm init -y

2. **Install Redux and Parcel**

   npm install redux

   npm install --save-dev parcel

3. **create index.html**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Redux</title>
</head>
<body>
  <h1>Redux Multiple Reducers Example</h1>
  <div>
        <h3>User Authentication Status:</h3>
        <p id="authStatus">Not Authenticated</p>
```

```
        <button id="loginButton">Login</button>
        <button id="logoutButton">Logout</button>
  </div>

  <div>
        <h3>Products:</h3>
        <ul id="productList"></ul>
        <button id="setProductsButton">Set Products</button>
  </div>
  <script src="script.js" type="module"></script>
</body>
</html>
```

4. create script.js

```
import { createStore, combineReducers } from 'redux';
// Reducer to manage authentication state
const authReducer = (state = { isAuthenticated: false }, action) => {
  switch (action.type) {
                case 'LOGIN':
                  return { ...state, isAuthenticated: true };
                case 'LOGOUT':
                  return { ...state, isAuthenticated: false };
                default:
                  return state;
  }};
// Reducer to manage product state
const productReducer = (state = { products: [] }, action) => {
  switch (action.type) {
                case 'SET_PRODUCTS':
                  return { ...state, products: action.payload };
                default:
                  return state;
  }};
// Combining both reducers into a root reducer
const rootReducer = combineReducers({
                        auth: authReducer,
                        product: productReducer, });
```

```
// Create a Redux store
const store = createStore(rootReducer,
__REDUX_DEVTOOLS_EXTENSION__()
);
// UI Update function to sync the UI with Redux state
const updateUI = () => {
        // Update Authentication Status
        const authState = store.getState().auth;
        document.getElementById('authStatus').textContent =
       authState.isAuthenticated ? 'Authenticated' : 'Not Authenticated';
        // Update Product List
        const productState = store.getState().product;
        const productList = document.getElementById('productList');
        productList.innerHTML = '';
        productState.products.forEach(product => {
                const listItem = document.createElement('li');
                listItem.textContent = product;
                productList.appendChild(listItem);
        });
};
// Dispatch actions based on button clicks
document.getElementById('loginButton').addEventListener('click', () => {
                store.dispatch({ type: 'LOGIN' });
                updateUI();
});
document.getElementById('logoutButton').addEventListener('click', () => {
                store.dispatch({ type: 'LOGOUT' });
                updateUI();
});
document.getElementById('setProductsButton').addEventListener('click',   ()
=> {
                store.dispatch({ type: 'SET_PRODUCTS', payload: ['HP
               Laptop', 'Wireless Mouse', 'Tablet'] });
                updateUI();
});
// Initialize UI
updateUI();
```

5. **Add some changes in package.json file**

```
"scripts": {
   "test": "echo \"Error: no test specified\" && exit 1",
   "start": "parcel index.html",
   "build": "parcel build index.html"
 }
```

6. **Run Parcel project:**

```
npm start
```

Another Example

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>  <meta charset="UTF-8">
 <meta      name="viewport"      content="width=device-width,      initial-
scale=1.0">
 <title>Redux</title>
</head>
<body>
 <h1>Redux combineReducers Example</h1>
 <div>
        <h3>User Status:</h3>
           <p id="userStatus">Not Logged In</p>
           <button id="loginButton">Login</button>
           <button id="logoutButton">Logout</button>
 </div>
 <div>
           <h3>Posts:</h3>
           <ul id="postList"></ul>
           <button id="fetchPostsButton">Fetch Posts</button>
 </div>
 <script src="script.js" type="module"></script>
</body>
</html>
```

script.js

```
import { createStore, combineReducers } from 'redux';
// user reducer
const initialUserState = {
                                isAuthenticated: false,
                                userDetails: null,
};
const userReducer = (state = initialUserState, action) => {
  switch (action.type) {
                case 'LOGIN':
                  return {
                                ...state,
                                isAuthenticated: true,
                                userDetails: action.payload
                  };
                case 'LOGOUT':
                  return {
                                ...state,
                                isAuthenticated: false,
                                userDetails: null
                  };
                default:
                  return state;
  }
};
// post reducer
const initialPostState = {
                            posts: [],
};

const postReducer = (state = initialPostState, action) => {
  switch (action.type) {
                        case 'FETCH_POSTS':
                          return {
                                ...state,
                                posts: action.payload
                          };
```

```
                        default:
                          return state;
  }
};
// combining reducers
const rootReducer = combineReducers({
                                                user: userReducer,
                                                posts: postReducer,
});

// creating the store
const store = createStore(rootReducer,
__REDUX_DEVTOOLS_EXTENSION__());

// updating the UI
const updateUI = () => {
                  // user status
                   const userState = store.getState().user;
                   const userStatusElement =
                 document.getElementById('userStatus');

                  if (userState.isAuthenticated) {
                     userStatusElement.textContent = `Logged In as
                 ${userState.userDetails.name} with email :
                 ${userState.userDetails.email}`;
                   }
                 else {
                     userStatusElement.textContent = 'Logged Out
                 Successfully';
                   }

                   // post list
                   const postState = store.getState().posts;
                   const postListElement =
                 document.getElementById('postList');
                   postListElement.innerHTML = '';
```

```
        postState.posts.forEach(post => {
          const listItem = document.createElement('li');
          listItem.textContent = post.title;
          postListElement.appendChild(listItem);
        });
      };

// dispatch actions based on button clicks
document.getElementById('loginButton').addEventListener('click', () => {
  const user = { name: 'Learn2Earn Labs', email:
'learn2earnlabs@gmail.com' };
  store.dispatch({ type: 'LOGIN', payload: user });
  updateUI();
});

document.getElementById('logoutButton').addEventListener('click', () => {
  store.dispatch({ type: 'LOGOUT' });
  updateUI();
});

document.getElementById('fetchPostsButton').addEventListener('click', ()
=> {
  const posts = [
    { id: 1, title: 'MERN Stack Training' },
    { id: 2, title: 'Java Full Stack Training' },
    { id: 3, title: 'Full Stack Software Engineer Training' },
    { id: 4, title: 'Cloud Computing & DevOps Training' },
    { id: 5, title: 'Digital Marketing & Business Development' },
  ];
  store.dispatch({ type: 'FETCH_POSTS', payload: posts });
  updateUI();
});

// initialize ui
updateUI();
```

Advantages of Using Multiple Reducers

- **Separation of Concerns:** Each reducer is responsible for a specific part of the application's state. This separation makes code more modular, easier to maintain, and less error-prone.
- **Scalability:** As the application grows, splitting the logic into multiple reducers allows better organization and scalability. New features can be added by creating additional reducers without modifying existing ones significantly.
- **Improved Readability and Debugging:** Smaller, focused reducers are easier to read and debug. Developers can pinpoint issues faster since each reducer manages only a portion of the state.
- **Testing:** Testing individual reducers becomes simpler when they are small and focused. This modular approach enables unit testing for each reducer independently.
- **Performance Optimization:** When you split the reducers, only the relevant reducer updates the part of the state it manages. This reduces unnecessary re-renders in components, potentially improving performance.

Use Cases for Multiple Reducers

- **Large Applications:** Applications with multiple features (e.g., authentication, products, orders, user profiles) benefit from separating the state into multiple reducers to manage each feature's state independently.
- **State Segmentation:** If different features or modules of your app deal with completely different pieces of data (e.g., a shopping cart system might have separate reducers for user data, cart items, and product listings).
- **Modular Features:** When developing an app with modular features that can be easily added or removed, splitting reducers allows modularity and independent development of each feature.
- **Team Collaboration:** In a team setting, multiple developers can work on different parts of the state without interfering with each other, as each can be responsible for their own reducer.

Connecting Redux with React

Redux is a state management library that allows you to manage the state of your application in a predictable and centralized way. When used with React, Redux provides a way to store global state outside of React components, making it easier to manage and share state across the app.

Step 1: Setting up a React and Redux Project

- Create a new React project using create-react-app

  npx create-react-app my-react-redux-app

  cd my-react-redux-app
- Install Redux and React-Redux

  npm install redux react-redux

  where,
  - Redux is the core library for managing global state.
  - React-Redux is a library that allows React components to interact with the Redux store.

Step 2: Create Redux Actions

Actions in Redux are plain JavaScript objects that describe what should be done to the state. Typically, actions have a type property, and sometimes a payload that contains the data to be processed.

src/redux/actions.js

```
export const increment = () => {
  return {
    type: 'INCREMENT',
  };
};

export const decrement = () => {
  return {
    type: 'DECREMENT',
  };
};
```

Step 3: Create Redux Reducers

Reducers are functions that take the current state and an action as arguments, and return a new state based on the action type.

src/redux/reducer.js

```
const initialState = {
                        count: 0,
};
const counterReducer = (state = initialState, action) => {
  switch (action.type) {
                          case 'INCREMENT':
                           return {
                                          ...state,
                                          count: state.count + 1,
                           };
                          case 'DECREMENT':
                           return {
                                          ...state,
                                          count: state.count - 1,
                           };
                          default:
                           return state;
  }
};
export default counterReducer;
```

Step 4: Create the Redux Store

The store holds the entire state tree of the application. You need to use the createStore function from Redux to create the store and provide the reducer to it.

src/redux/store.js

```
import { createStore } from 'redux';
import counterReducer from './reducer';
const store = createStore(counterReducer);
export default store;
```

Step 5: Connect Redux to React

To connect Redux with React, you need to wrap your application with the Provider component from react-redux and pass the Redux store to it.

- **Using Redux State and Dispatch in React Components**

  In your React component, you will use the useSelector hook to access the Redux state and the useDispatch hook to dispatch actions.

  src/App.js

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement } from './redux/actions';
function App() {
 // Accessing state from the Redux store
 const count = useSelector(state => state.count);
 const dispatch = useDispatch();
 return (
  <div className="App">
   <h1>Count: {count}</h1>
   <button onClick={() => dispatch(increment())}>Increment</button>
   <button onClick={() => dispatch(decrement())}>Decrement</button>
  </div>
 );
}
export default App;
```

- **Wrap your app in the Provider**

  src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import App from './App';
import store from './redux/store';
ReactDOM.render(
        <Provider store={store}>  <App />
        </Provider>,
        document.getElementById('root')
);
```

Best Practices and Standards for Using Redux in React

- **Organize Redux Code Properly:** Keep your Redux code organized by separating it into folders such as actions, reducers, store, and types (for action types).

```
src/
  redux/
    actions/
      counterActions.js
    reducers/
      counterReducer.js
    store.js
  components/
    CounterComponent.js
```

- **Use Action Constants:** Define action types as constants to avoid typos in action names and make debugging easier.

Example:

```
export const INCREMENT = 'INCREMENT';
export const DECREMENT = 'DECREMENT';
```

- **Avoid Mutating State in Reducers:** Always return a new object in the reducer and never mutate the original state. Use techniques like the spread operator (...) to create copies of objects or arrays.

Example:

```
return {
  ...state,
  count: state.count + 1
};
```

- **Use combineReducers for Scalability:** If your app has multiple state slices (e.g., user, posts, products), use combineReducers to split the reducers into manageable pieces.

Example:

```
import { combineReducers } from 'redux';
const rootReducer = combineReducers({
  counter: counterReducer,
  user: userReducer,
});
```

- **Avoid Redundant State:** Only store necessary information in the Redux state. Derived data that can be computed from existing state (like filtered or sorted lists) should not be stored in the Redux store.

- **Normalize Complex State:** If you're managing deeply nested or relational data, normalize the state shape. Use libraries like normalizr to flatten nested data and store it in a more efficient way.
  Example state normalization:

```
{
  "users": {
        "1": { "id": 1, "name": "Neha Rathore" },
        "2": { "id": 2, "name": "Tushar Gupta" }
  },
  "posts": {
        "101": { "id": 101, "title": "Post 1", "userId": 1 },
        "102": { "id": 102, "title": "Post 2", "userId": 2 }
  }
}
```

- **Use Middleware for Async Actions:** Use redux-thunk or redux-saga for handling side effects (like API calls) in Redux. This ensures that the logic for asynchronous actions is separated from the reducers.
  Example of using redux-thunk:

```
import thunk from 'redux-thunk';
const store = createStore(rootReducer, applyMiddleware(thunk));
```

- **Use DevTools for Debugging:** Integrate Redux DevTools for easier debugging and state inspection.
  Example:

```
import { composeWithDevTools } from 'redux-devtools-extension';
const store = createStore(rootReducer, composeWithDevTools());
```

- **Keep Components Dumb, Use Selectors:** Use selectors (functions that extract data from the state) to keep the components "dumb" and focused only on rendering.
  Example:

```
const getCount = state => state.counter.count;
const count = useSelector(getCount);
```

Example : React Todo Application with Redux

1. **Create a React project using create-react-app**

   npx create-react-app my-redux-todo-app
   cd my-redux-todo-app

2. **Install Redux and React-Redux**

   npm install redux react-redux

3. **Create Redux Actions**

   We need actions to handle adding, deleting, and toggling the completion of tasks.

   src/redux/actions/todoActions.js

```
export const addTodo = (todo) => {
                return {
                                type: 'ADD_TODO',
                                payload: todo,
                };
};

export const deleteTodo = (id) => {
 return {
                                type: 'DELETE_TODO',
                                payload: id,
 };
};

export const toggleTodo = (id) => {
 return {
                                type: 'TOGGLE_TODO',
                                payload: id,
 };
};
```

4. **Create Redux Reducer**

The reducer will handle the logic for adding, deleting, and toggling the tasks based on the action type. Each task will have a completed property to track if it's done.

src/redux/reducers/todoReducer.js

```
const initialState = {
        todos: [],
};

const todoReducer = (state = initialState, action) => {
    switch (action.type) {
        case 'ADD_TODO':
          return {
            ...state,
            todos: [...state.todos, { id: Date.now(), task: action.payload, completed: false }],
          };
        case 'DELETE_TODO':
          return {
            ...state,
            todos: state.todos.filter((todo) => todo.id !== action.payload),
          };
        case 'TOGGLE_TODO':
          return {
            ...state,
            todos: state.todos.map((todo) =>
              todo.id === action.payload ? { ...todo, completed: !todo.completed } : todo
            ),
          };
        default:
          return state;
    }
};
export default todoReducer;
```

5. **Create the React Component**

Now create a Todo component that interacts with the Redux store.

src/components/Todo.js

```
import React, { useState } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { addTodo, deleteTodo, toggleTodo } from
'../redux/actions/todoActions';

function Todo() {
 const [input, setInput] = useState('');
 const todos = useSelector((state) => state.todos);
 const dispatch = useDispatch();

 const handleSubmit = (e) => {
  e.preventDefault();
  if (input.trim()) {
   dispatch(addTodo(input));
   setInput('');
  }
 };

 return (
  <div>
   <h1>Todo List</h1>
   <form onSubmit={handleSubmit}>
    <input
     type="text"
     value={input}
     onChange={(e) => setInput(e.target.value)}
     placeholder="Add a task"
    />
    <button type="submit">Add Todo</button>
   </form>
   <ul>
    {todos.map((todo) => (
```

```
            <li key={todo.id} style={{ textDecoration: todo.completed ? 'line-
through' : 'none' }}>
              {todo.task}
              <button onClick={() => dispatch(toggleTodo(todo.id))}>
                {todo.completed ? 'Undo' : 'Complete'}
              </button>
              <button onClick={() =>
dispatch(deleteTodo(todo.id))}>Delete</button>
            </li>
          ))}
        </ul>
      </div>
    );
}

export default Todo;
```

6. **App Component**

In the App.js file, import the Todo component and render it.

src/App.js

```
import React from 'react';
import Todo from './components/Todo';

function App() {
  return (
    <div>
      <Todo />
    </div>
  );
}

export default App;
```

7. **Create the Redux Store**
   Combine your reducer with the store so that the app can access the state.

   src/redux/store.js

```
import { createStore } from 'redux';
import todoReducer from './reducers/todoReducer';
const store = createStore(todoReducer);
export default store;
```

8. **Connect Redux to React**
   To allow your React components to use Redux state and dispatch actions, you need to wrap your app in the Provider component from react-redux.

   src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import App from './App';
import store from './redux/store';

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

9. **Run the Application**
   You can now start your application with:

```
npm start
```

Conditional DevTools Setup

Conditional DevTools Setup refers to configuring Redux DevTools so that it is only used in certain environments, typically in development. This approach prevents the inclusion of the DevTools extension in production builds, where it is not necessary and might even expose sensitive data or add unnecessary overhead.

Purpose of Conditional DevTools Setup

- **Development vs. Production:** In development environments, Redux DevTools can be extremely helpful for debugging and inspecting the state and actions of your Redux store. However, in production environments, DevTools are not needed and might pose security risks or performance overhead.
- **Avoiding Unnecessary Dependencies:** By conditionally adding DevTools, you ensure that only necessary dependencies and configurations are included in your production builds, which helps keep the build clean and efficient.

How to Implement Conditional DevTools Setup

You can conditionally integrate Redux DevTools using environment variables or by checking if the DevTools extension is available in the browser.

src/redux/store.js

```
import { createStore } from 'redux';
import todoReducer from './reducers/todoReducer';

const devTools =
  window.__REDUX_DEVTOOLS_EXTENSION__ &&
window.__REDUX_DEVTOOLS_EXTENSION__();

// Create the Redux store, conditionally add DevTools only in development
const store = createStore(
  todoReducer,
  process.env.NODE_ENV === 'development' ? devTools : undefined
);

export default store;
```

- **In Development**
    - process.env.NODE_ENV is usually set to 'development'. The composeWithDevTools() function is used, which enables Redux DevTools in the browser.
    - It checks if window.__REDUX_DEVTOOLS_EXTENSION__ is available. If so, it uses it to integrate with Redux DevTools.
- **In Production**
    - process.env.NODE_ENV is set to 'production', and undefined is passed instead. This prevents DevTools from being included in the production build.
    - It does not use the DevTools extension, avoiding unnecessary configuration and potential risks.

Benefits of Conditional DevTools Setup

- **Security:** Avoids exposing potentially sensitive state information in production.
- **Performance:** Reduces overhead and improves performance by excluding unnecessary debugging tools from the production build.
- **Cleaner Builds:** Ensures that production builds are free of development-specific tools and configurations.

How to Check Environment

- **Create React App:** Automatically sets process.env.NODE_ENV to 'development' or 'production' based on the build command (npm start or npm run build).
- **Custom Configurations:** You can set up environment variables using .env files or custom configurations if you're using other build tools.