We will refactor the todos list from Lab 6 into Redux.

To get it working, you have to install:

@reduxjs/toolkit: This is the official, opinionated, batteries-included toolset for efficient Redux development. It simplifies common tasks like setting up the store, creating reducers, and handling immutable updates.

react-redux: This library provides bindings to connect your React components to the Redux store. It includes the `useSelector` and `useDispatch` hooks that we use in the code.

REDUX CONCEPTS:

1. **Store:** Think of the store as a JavaScript object that holds the entire state of your application. In this case, the state is the `todos` array, which represents the list of tasks. The store is created using `configureStore` from Redux Toolkit, and it takes the `reducer` from the slice as its argument.

2. **Slice:** A slice is a way to organize a part of your Redux store. It bundles together the state, actions, and reducers related to a specific feature. The `todosSlice` manages the state and logic for our to-do list.

3. **Reducers:** Reducers are pure functions that take the previous state and an action as arguments, and they return a new state. They define how the state should change in response to different actions. In our case, `addTodo` adds a new to-do item to the state array, and `removeTodo` filters out the to-do item with the matching ID.

4. **Actions:** Actions are plain JavaScript objects that describe events that occurred in the application. They have a `type` property (which is a string) and can optionally have a `payload` property to carry additional data. The actions are dispatched from the components to the store. For example, `dispatch(addTodo(todoText))` means "dispatch the `addTodo` action with the text of the new to-do item as the payload."

5. **Selectors:** Selectors are functions that take the entire state as input and return a specific piece of state that a component needs. The `useSelector` hook from `react-redux` allows you to access the state within your components. In the code, `useSelector(state => state)` selects the entire state, which is the `todos` array in our case.

How the Code Works:

1. **Store Creation (store.js):**

- The `todosSlice` is created, defining the initial state (empty array), reducers (`addTodo` and `removeTodo`), and actions (also `addTodo` and `removeTodo`).
- The store is configured using `configureStore`, taking the reducer from the slice.

2. **TodoApp Component (TodoApp.jsx):**

   - **useSelector:** This hook connects the component to the Redux store and retrieves the `todos` state. Whenever the state changes, the component re-renders to reflect the updates.
   - **useDispatch:** This hook gives you a function called `dispatch`. When you call `dispatch(action)`, it sends the action to the Redux store, which then updates the state based on the reducer logic.
   - **Input Handling:** A ref (`inputRef`) is used to get a reference to the input element. When the "Add Todo" button is clicked or Enter is pressed in the input field, the `handleAddTodo` function is called.
   - **Adding a Todo:** The text from the input is retrieved, trimmed, and if it's not empty, the `addTodo` action is dispatched with the text as the payload. The input is then cleared.
   - **Rendering the List:** The component iterates over the `todos` array and renders each to-do item as a list item (`<li>`). A delete button is also rendered for each item, and clicking it dispatches the `removeTodo` action with the item's ID.

3. **Main Entry Point (index.js):**

   - **Provider:** The `Provider` component from `react-redux` wraps the entire app. It takes the store as a prop and makes it available to all nested components. This means any component in the app can now use `useSelector` and `useDispatch` to interact with the store.

Why Redux? Redux offers several benefits over the class-based approach:

- **Centralized State:** State management is easier because the state is stored in a single location (the store).
- **Predictable State Changes:** The only way to change the state is by dispatching actions, which are then processed by reducers. This makes the app's behavior more predictable and easier to debug.
- **Component Reusability:** Components become more reusable because they don't rely on directly manipulating state within themselves.

Redux Store Setup (store.js):

JavaScript

```
import { configureStore, createSlice } from '@reduxjs/
toolkit';
```

```
const todosSlice = createSlice({
  name: 'todos',
  initialState: [],
  reducers: {
    addTodo: (state, action) => {
      state.push({ id: Date.now(), text: action.payload });
    },
    removeTodo: (state, action) => {
      return state.filter(todo => todo.id !==
action.payload);
    }
  }
});

export const { addTodo, removeTodo } = todosSlice.actions;
export default configureStore({ reducer: todosSlice.reducer
});
```

- **Slice:** This encapsulates the state (todos array) and the reducers (addTodo and removeTodo) for this part of the application.
- **Actions:** The slice automatically creates actions for us based on the reducer names.
- **Store:** This holds the entire state tree of your application.

2. React Component (TodoApp.jsx):

JavaScript

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { addTodo, removeTodo } from './store'; // Assuming
store.js is in the same directory

const TodoApp = () => {
  const todos = useSelector(state => state); // Get the
todos state
  const dispatch = useDispatch();
  const inputRef = React.useRef(null);

  const handleAddTodo = () => {
    const todoText = inputRef.current.value.trim();
    if (todoText) {
      dispatch(addTodo(todoText));
      inputRef.current.value = '';
```

```
    }
  };

  return (
    <div>
      <input ref={inputRef} id="todo-input" type="text" />
      <button onClick={handleAddTodo} id="add-btn">Add
Todo</button>

      <ul id="todo-list">
        {todos.map(todo => (
          <li key={todo.id} className="todo-item">
            <span>{todo.text}</span>
            <button onClick={() =>
dispatch(removeTodo(todo.id))}>Delete</button>
          </li>
        ))}
      </ul>
    </div>
  );
};

export default TodoApp;
```

- **useSelector:** Connects the component to the Redux store and retrieves the `todos` data.
- **useDispatch:** Gives the component the ability to dispatch actions to the Redux store.
- **React Hooks:** Used to manage input value and reference.

3. Main Entry Point (index.js):

JavaScript

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

const root =
ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <Provider store={store}>
    <TodoApp />
```

```
  </Provider>
);
```

- **Provider:** This makes the Redux store available to all nested components in your React application.

Key Improvements:

- **Centralized State Management:** Redux helps you manage the `todos` array in a single place.
- **Predictable State Changes:** Changes to state are made through pure functions (reducers), making it easier to reason about and debug.
- **Improved Component Structure:** The React component becomes more focused on UI logic, and event handlers dispatch actions to the store instead of directly modifying state.