Redux Example :

1. Setting Up the Redux Store

In Redux, the store holds the entire state of your application. It's created using the `createStore()` function from Redux.

```javascript
// store.js
import { createStore } from 'redux';
import rootReducer from './reducers';

const store = createStore(rootReducer);

export default store;
```

In this example, we import `createStore` from Redux and combine our reducers using `combineReducers` if there are multiple reducers. Then, we create the Redux store using `createStore` and pass in the root reducer.

2. Defining Reducers

Reducers specify how the application's state changes in response to actions sent to the store. They are pure functions that take the previous state and an action, and return the next state.

```javascript
// reducers/todoReducer.js
const initialState = {
  todos: []
};

const todoReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'ADD_TODO':
      return {
        ...state,
        todos: [...state.todos, action.payload]
      };
    case 'REMOVE_TODO':
      return {
        ...state,
        todos: state.todos.filter(todo => todo.id !== action.payload)
      };
```

```
    default:
      return state;
  }
};

export default todoReducer;
```

In this reducer example, we define an initial state with an empty array of todos. We handle two actions: `ADD_TODO` to add a todo to the list, and `REMOVE_TODO` to remove a todo from the list.

3. Creating Action Creators

Action creators are functions that create actions to update the state. They return action objects with a `type` and `payload`.

```javascript
// actions/todoActions.js
export const addTodo = (todo) => {
  return { type: 'ADD_TODO', payload: todo };
};

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

These action creators are called from React components to dispatch actions to the Redux store.

4. Creating React Components

React components interact with the Redux store by dispatching actions and reading state from the store. They are typically connected to the store using the `connect()` function from `react-redux`.

Example Component: TodoList

```javascript
// components/TodoList.js
import React from 'react';
import { connect } from 'react-redux';
import { removeTodo } from '../actions/todoActions';
```

```
const TodoList = ({ todos, dispatch }) => {
  return (
    <ul>
      {todos.map(todo => (
        <li key={todo.id}>
          {todo.text}
          <button onClick={() => dispatch(removeTodo(todo.id))}>Remove</button>
        </li>
      ))}
    </ul>
  );
};

const mapStateToProps = (state) => {
  return {
    todos: state.todos.todos
  };
};

export default connect(mapStateToProps)(TodoList);
```

In this example, `TodoList` is connected to the Redux store using `connect()`. It receives todos as props from the Redux store and dispatches `REMOVE_TODO` actions when the remove button is clicked.

5. Creating the App Component

The main `App` component renders other React components and serves as the entry point of the application.

```javascript
// App.js
import React from 'react';
import TodoList from './components/TodoList';

const App = () => {
  return (
    <div>
      <h1>Todo List</h1>
      <TodoList />
    </div>
  );
};
```

```
export default App;
```

6. Rendering the App

Finally, we render the `App` component inside the `Provider` component from `react-redux`, which wraps the entire application and provides access to the Redux store.

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

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

This setup allows us to manage the state of our React application using Redux. Actions are dispatched to the store from React components, and reducers update the state accordingly. The UI re-renders automatically whenever the state changes.