Redux

Redux is a JavaScript library for managing the state of web applications. It provides a centralized store for the state, allowing for more predictable and manageable behaviour, particularly in large and complex applications with frequent changes to the state. It is often used with React, but can also be used with other JavaScript frameworks or libraries.

Redux in React Js

In React, Redux is a tool that helps to manage the state of a web application. It allows you to store the state of your application in one central place, which makes it easier to manage and less prone to errors. This central store can be accessed by all the components in your application, which simplifies how they can interact with each other. Using Redux with React can help make your web application more reliable and easier to maintain.

Advantages using Redux with React js

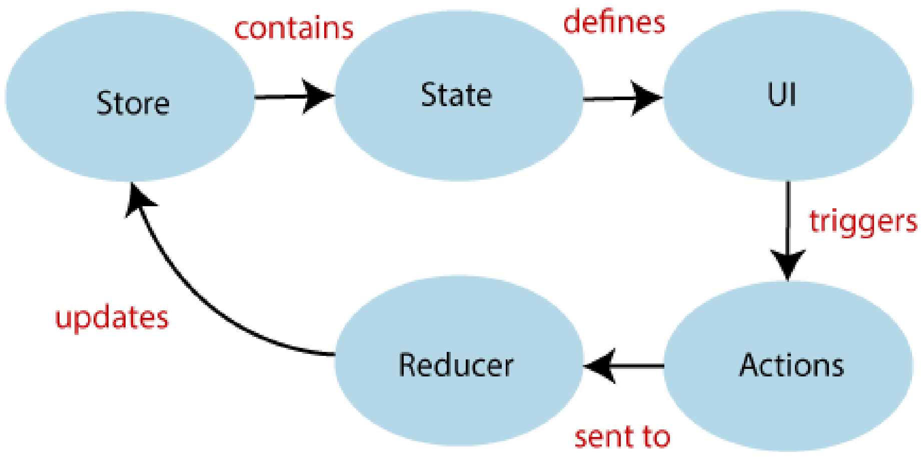
Here are the advantages of using Redux with React in simpler language:

1. One place to manage state: With Redux, you can store the state of your application in one central place, which makes it easier to manage and update.
2. Clear and consistent state updates: Redux gives you a way to update the state of your application in a consistent way, which makes it easier to understand and fix problems with your code.
3. Easy debugging: Redux keeps a record of the state of your application over time, so you can go back and see how it changed at different times. This makes it easier to find and fix problems with your code.
4. Works with React and other libraries: Redux can be used with React and other JavaScript libraries, which makes it easier to integrate into your projects.
5. Makes large projects more manageable: As your React application grows, Redux can help you manage the complexity of the code by providing a consistent way to handle state changes.

Different parts of react redux application

1. **Store**: The store is like a central container that holds all the data of your application. It keeps track of the state of the application and provides a way to update and retrieve the state.
2. **Actions**: Actions are like messages that describe something that happened in your application. They are used to trigger updates to the state of the application.
3. **Reducers**: Reducers are like functions that listen for actions and use them to update the state of the application.
4. **Dispatch**: Dispatch is like a messenger that sends actions to the reducers. When an action is dispatched, it triggers a state update.
5. **Provider**: Provider is like a helper component from the React-Redux library that helps to provide the store to all the components in the application.
6. **Connect**: Connect is like a function from the React-Redux library that connects a component to the store. It provides access to the state and actions that can update the state.

These are the main parts of a React-Redux application. The store holds the data, actions describe what happened, reducers update the data, dispatch sends actions to reducers, Provider helps to provide the store to all components, and Connect helps to connect a component to the store.



Installing redux to the react js application
To install the latest version of Redux and the recommended libraries, follow these steps:

1. Open your terminal and navigate to the root directory of your React application.
2. Install the latest version of Redux and the recommended libraries using npm (the Node package manager) by running the following command:
   npm install --save redux react-redux @reduxjs/toolkit
   This will install the latest version of Redux, React-Redux, and the Redux toolkit, and add them to your package.json file.
3. In your application's code, you can import the libraries as follows:
   import { configureStore } from '@reduxjs/toolkit';
   import { Provider } from 'react-redux';

    1. **configureStore** is a function that creates a Redux store for your React application. It sets up the store with some good defaults and features that make it easier to use, like making sure the data in the store can't be accidentally changed, and making it easy to store and retrieve the state of the application.
    2. **Provider** is a component from the React-Redux library that makes the Redux store available to all the components in your React application. It allows you to "provide" the store to your app, so that any component can access the state in the store and update it as necessary.

       Together, configureStore and Provider make it easier to manage the state of your React application with Redux by providing a simple way to create and access the Redux store.
4. Create a Redux store in your application's code using the configureStore function. The configureStore function takes an object that defines the initial state and the reducers that will handle state updates based on the actions that are dispatched to the store.
    1. In a React-Redux application, you can use the **configureStore** function provided by the Redux toolkit to create a Redux store.

2. **configureStore** takes an object that has two properties: **reducer** and **preloadedState**.
3. The **reducer** property is a function that handles state updates based on actions that are dispatched to the store. It should return a new state based on the current state and the action that was dispatched.
4. The **preloadedState** property is an optional argument that lets you specify the initial state of the store. If you don't specify it, the store will be initialized with the default value specified in your reducer function.
5. Once you have defined the **reducer** and **preloadedState** properties, you can pass the object to the **configureStore** function to create your Redux store.
6. The **configureStore** function will return a new Redux store, which you can use to manage the state of your React application.

```
import { configureStore } from '@reduxjs/toolkit';
const initialState = { count: 0 };

function 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;
 }
}
const store = configureStore({ reducer: counterReducer });
```

5. Wrap your React application in a **Provider** component and pass in the Redux store as a prop. This makes the store available to all of your application's components. Here's an example:

```
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')
);
```

In this example, we wrap our **App** component in a **Provider** component and pass in the Redux store as a prop.

That's it! With these steps, you have installed the latest version of Redux with a React application and set up a Redux store to manage the application's state, using the latest and recommended libraries.

Simple React JS redux application.

1. Create a new React application using **create-react-app**. You can do this by opening your terminal and running the following command:
   ```
   npx create-react-app my-app
   ```
   This will create a new React application in a directory called **my-app**.
2. In the root directory of your React application, install the latest version of Redux and React-Redux using the following command:
   ```
   npm install  redux react-redux @reduxjs/toolkit
   ```
   This will install the latest version of Redux, React-Redux, and the Redux toolkit, and add them to your package.json file.
3. In the **src** directory of your React application, create a new file called **store.js**. In this file, you can use the **configureStore** function to create a Redux store with a simple reducer function:
   ```
   import { configureStore } from '@reduxjs/toolkit';

   const initialState = { count: 0 };

   function 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;
     }
   }

   const store = configureStore({ reducer: counterReducer });

   export default store;
   ```
   In this example, we define an initial state of **{ count: 0 }**, a reducer function **counterReducer** that increments or decrements the count based on the type of action that is dispatched, and then use the **configureStore** function to create the Redux store.
4. In the **src** directory of your React application, create a new file called **App.js**. In this file, you can define a simple React component that displays the current count and provides buttons to increment or decrement the count:
   ```
   import React from 'react';
   import { useSelector, useDispatch } from 'react-redux';

   function App() {
   ```

```
  const count = useSelector(state => state.count);
  const dispatch = useDispatch();

  const increment = () => {
    dispatch({ type: 'INCREMENT' });
  };

  const decrement = () => {
    dispatch({ type: 'DECREMENT' });
  };

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
}

export default App;
```

In this example, we use the **useSelector** and **useDispatch** hooks from React-Redux to select the current count from the store and dispatch actions to update the count.

5. In the **src** directory of your React application, create a new file called **index.js**. In this file, you can use the **Provider** component from React-Redux to provide the store to your application:

```
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')
);
```

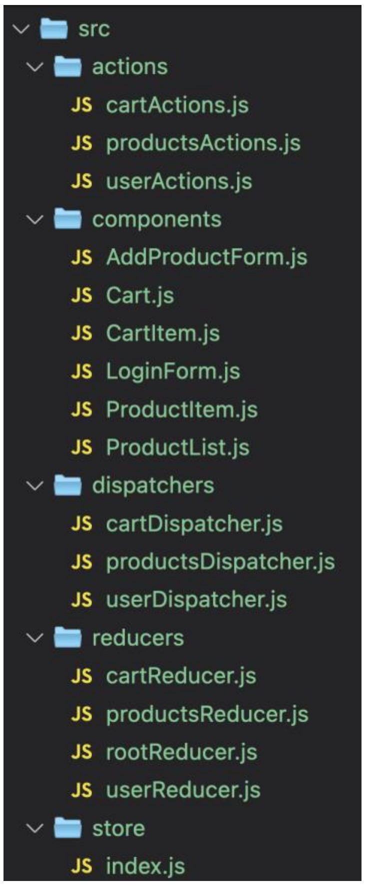
In this example, we wrap our **App** component in a **Provider** component and pass in the Redux store as a prop.

That's it! With these steps, you have created a very simple React-Redux application that uses optimised code

eCommerce application on react redux using multiple reducers, actions and dispatchers.

1. Create a new React application using **create-react-app**. You can do this by opening your terminal and running the following command:
   npx create-react-app my-app
   This will create a new React application in a directory called **my-app**.
2. In the root directory of your React application, install the latest version of Redux and React-Redux using the following command:
   npm install --save redux react-redux @reduxjs/toolkit react-hook-form
   npm install react-router-dom bootstrap
   This will install the latest version of Redux, React-Redux, and the Redux toolkit, and add them to your package.json file.
3. In the **src** directory of your React application, create four new folders: **store**, **reducers**, **dispatchers**, and **actions**. In the **reducers** folder, create three new files called **productsReducer.js**, **cartReducer.js**, and **userReducer.js**. In the **dispatchers** folder, create three new files called **productsDispatcher.js**, **cartDispatcher.js**, and **userDispatcher.js**. In the **actions** folder, create three new files called **productsActions.js**, **cartActions.js**, and **userActions.js**. In the **store** folder, create a new file called **index.js**. In the components folder, create three new files called **Cart.js**, **CartItem.js**, **ProductList.js**, **ProductItem.js**, **LoginForm.js** and **AddProductForm.js**. Your directory structure should look like this:

```
src/
  store/
    index.js
  reducers/
    productsReducer.js
    cartReducer.js
    userReducer.js
  dispatchers/
    productsDispatcher.js
    cartDispatcher.js
    userDispatcher.js
  actions/
    productsActions.js
    cartActions.js
    userActions.js
  components/
    Cart.js
    CartItem.js
    ProductList.js
    ProductItem.js
    LoginForm.js
    AddProductForm.js
```



```
src
  actions
    cartActions.js
    productsActions.js
    userActions.js
  components
    AddProductForm.js
    Cart.js
    CartItem.js
    LoginForm.js
    ProductItem.js
    ProductList.js
  dispatchers
    cartDispatcher.js
    productsDispatcher.js
    userDispatcher.js
  reducers
    cartReducer.js
    productsReducer.js
    rootReducer.js
    userReducer.js
  store
    index.js
```

4. Add following code for all reducers like following
    1. **reducers/productsReducer.js**

```
const initialState = [];
const productsReducer = (state = [], action) => {
  switch (action.type) {
    case 'ADD_PRODUCT':
      const { name, price } = action.payload;
      return [...state, {['id']:state.length+1,name,price}];
    case 'REMOVE_PRODUCT':
      return state.filter(product => product.id !== action.payload);
      case 'UPDATE_PRODUCT':
        const { payload } = action;
        let obj=state.map(product => product.id === payload.id ? payload :
       product);
        return state.map(product => product.id === payload.id ? payload :
       product);
    default:
      return state;
  }
};
export default productsReducer;
```

    2. **reducers/cartReducer.js**

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

const cartReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'ADD_TO_CART': {
      const newItem = action.payload;
      const existingItem = state.items.find(item => item.id === newItem.id);
      if (existingItem) {
        return {
          ...state,
          items: state.items.map(item =>
            item.id === newItem.id ? { ...item, quantity: item.quantity + 1 } : item
          ),
        };
      } else {
        return {
          ...state,
          items: [...state.items, { ...newItem, quantity: 1 }]
        };
      }
    }
    case 'REMOVE_FROM_CART': {
```

```
    const cartItem = action.payload;
      ...state,
      items: state.items.filter(item => item.id !== cartItem.id),
    };
  }
  case 'EMPTY_CART': {
    return {
      ...state,
      items: [],
    };
  }
  default:
    return state;
 }
};
export default cartReducer;
```

3. **reducers/userReducer.js**

```
const initialState = {
  name: '',
  email: '',
  isLoggedIn: false,
};
 const userReducer = (state = initialState, action) => {
  switch (action.type) {
   case 'LOGIN':
     return {
       name: action.payload.name,
       email: action.payload.email,
       isLoggedIn: true,
     };
   case 'LOGOUT':
     return initialState;
   default:
     return state;
  }
};
export default userReducer;
```

4. **reducers/rootReducers.js**

```
import { combineReducers } from 'redux';
import cartReducer from './cartReducer';
import userReducer from './userReducer';
import productsReducer from './productsReducer';

const rootReducer = combineReducers({
  cart: cartReducer,
  products: productsReducer,
  user: userReducer,
});
export default rootReducer;
```

6. Add following code for all actions as follows
   1. **actions/productsActions.js**

      ```
      export const addProduct = (product) => ({ type: 'ADD_PRODUCT', payload: product });
      export const deleteProduct = (product) => ({ type: 'DELETE_PRODUCT', payload: product });
      export const updateProduct = (product) => ({ type: 'UPDATE_PRODUCT', payload: product });
      ```

   2. **actions/cartActions.js**

      ```
      export const addToCart = (product) => ({ type: 'ADD_TO_CART', payload: product });
      export const removeFromCart = (product) => ({ type: 'REMOVE_FROM_CART', payload: product });
      export const emptyCart = () => ({ type: 'EMPTY_CART' });
      ```

   3. **actions/userActions.js**

      ```
      export const login = (user) => ({
          type: 'LOGIN',
          payload: user,
        });

        export const logout = () => ({
          type: 'LOGOUT',
        });
      ```

7. Add following code for all dispatchers as follows
   1. **dispatchers/productsDispatcher.js**

      ```
      import { addProduct, removeProduct } from '../actions/productsActions';
      export function addNewProduct(dispatch, product) {
        dispatch(addProduct(product));
      }

      export function removeExistingProduct(dispatch, product) {
        dispatch(removeProduct(product));
      }
      ```

   2. **dispatchers/cartDispatcher.js**

      ```
      import { addToCart, removeFromCart, emptyCart } from '../actions/cartActions';
      export function addItemToCart(dispatch, item) {
        dispatch(addToCart(item));
      }

      export function removeItemFromCart(dispatch, item) {
        dispatch(removeFromCart(item));
      ```

```
}

export function emptyCartItems(dispatch) {
  dispatch(emptyCart());
}
export function emptyCartItems(dispatch) {
  dispatch(emptyCart());
}
```

3. **dispatchers/userDispatcher.js**

```
import { login } from '../actions/userActions';

export const loginUser = (email, password) => {
  return (dispatch) => {
    // Make API call to authenticate user
    // On success, dispatch the login action with the user's details
    dispatch(login({ email, name: 'John Doe' }));
  };
};
```

8. create **index.js** inside **store/index.js**

```
import { configureStore } from '@reduxjs/toolkit';
import productsReducer from './reducers/productsReducer';
import cartReducer from './reducers/cartReducer';
import userReducer from './reducers/userReducer';

const store = configureStore({
  reducer: {
    products: productsReducer,
    cart: cartReducer,
    user: userReducer,
  },
});
export default store;
```

9. create following components from **src/components** folder
    1. **src/components/ AddProductForm**

```
import { useForm } from 'react-hook-form';
import { useDispatch, useSelector } from 'react-redux';
import { addProduct } from '../actions/productsActions';

function AddProductForm() {
  const { register, handleSubmit, formState: { errors } } = useForm();
  const dispatch = useDispatch();

  const products = useSelector(state => state.products);

  const onSubmit = data => {
    dispatch(addProduct(data));
```

```
 };

 return (
  <div>
  <form onSubmit={handleSubmit(onSubmit)}>
  <div>
  <input placeholder='Product Name'
     name='name'
     {...register('name',{required:true})}
     />
     <span>
     {errors.name && <span>Product Name is required</span>}
     </span>
  </div>
  <div>
  <input placeholder='Product Price'
     name='price'
     {...register('price',{required:true})}
  />
     <span>
     {errors.price && <span>Product Price is required</span>}
     </span>
  </div>

  <div>
  <textarea placeholder='Description'
     name='description'
     {...register('description',{required:true})}
  />
     <span>
     {errors.description && <span>Product Description is required</span>}
     </span>
  </div>
   <button type="submit">Add Product</button>
  </form>
  <hr/>
  {JSON.stringify(products)}
  </div>
 );
}
export default AddProductForm
```

2. **src/components/Cart.js**

```
import React from 'react';
import { useSelector } from 'react-redux';
import ProductItem from './ProductItem';

function ProductList() {
  const products = useSelector(state => state.products);
  return (
    <div>
      <h2>Products</h2>
      {products.length === 0 ? (
        <p>No products available.</p>
      ) : (
        <div>
          {products.map(product => (
            <ProductItem key={product.id} product={product} />
          ))}
        </div>
      )}
    </div>
  );
}

export default ProductList;
```

3. **src/components/ProductItems.js**

```
import React from 'react';
import { useDispatch, useSelector } from 'react-redux';
import { addToCart } from '../actions/cartActions';
import { updateProduct } from '../actions/productsActions';

function ProductItem({ product }) {
  const dispatch = useDispatch();

  const handleAddToCart = (e) => {
    const updatedProduct = { ...product, inCart: true };
    dispatch(addToCart(updatedProduct));
  };

  return (
    <div className='col-4'>
      <div className="card mt-2">
        <div className="card-header"> <h2>{product.name}</h2></div>
        <div className="card-body">
          <div className="card-text">
            <p>Description:{product.description}</p>
            <p>Price: {product.price}</p>
            <hr/>
            {!product.inCart && (
              <>
                <button onClick={handleAddToCart}>Add to Cart</button>
              </>
            )}
          </div>
        </div>
      </div>
    </div>
  );
}
export default ProductItem;
```

4. **src/components/ProductList.js**

```
import React from 'react';
import { useSelector } from 'react-redux';
import ProductItem from './ProductItem';

const ProductList=()=> {
  const products = useSelector(state => state.products); // gets from store
  return (
    <div>
      <h2>Products</h2>
      <hr/>
      {products.length === 0 ? (
        <p>No products available.</p>
      ) : (
        <div className='row'>
          {products.map(product => (
            <ProductItem key={product.id} product={product} />
          ))}
        </div>
      )}
    </div>
  );
}
export default ProductList;
```

5. **src/components/CartItem.js**

```
import React from 'react';
import { useDispatch } from 'react-redux';
import { removeFromCart } from '../actions/cartActions';

const CartItem = ({ product }) => {
    const dispatch = useDispatch();
    const handleRemoveFromCart = () => {
        dispatch(removeFromCart(product)); // dispatch to action
    };

    return (
        <tr>
            <td>{product.name}</td>
            <td>  {product.quantity}</td>
            <td>&#8377;{product.price}</td>
            <td>&#8377;{product.price * product.quantity}</td>
            <td><button
onClick={handleRemoveFromCart}>Remove</button></td>
```

```
        </tr>
    );
};
export default CartItem;
```

6. **src/components/Cart.js**

```
import React from 'react';
import { useSelector } from 'react-redux';
import { useDispatch } from 'react-redux';
import CartItem from './CartItem';
import { emptyCart, removeFromCart } from '../actions/cartActions';

const Cart = () => {
    const dispatch = useDispatch();

    const cartItems = useSelector(state => state.cart.items);
    const cartTotal = cartItems.reduce((total, item) => total + item.price *
item.quantity, 0);
    const handleRemoveFromCart = (product) => {
        dispatch(removeFromCart(product));
    };
    const handleClearCart = () => {
        dispatch(emptyCart());
     };

    return (
        <div>
            <div className='row'>
                <div className='col-5'>
                    <h2>Cart</h2>
                </div>
                <div className='col-7'>
                <button onClick={handleClearCart}>Clear Cart</button>
                </div>
            </div>
            <hr />
            {cartItems.length === 0 && <p>Your cart is empty.</p>}
            {cartItems.length > 0 &&
                <table className='table'>
                    <thead>
                        <tr>
                            <th>Product Name</th>
                            <th>Quantity</th>
                            <th>Unit Price</th>
                            <th>Total</th>
                            <th>Action</th>
                        </tr>
                    </thead>
                    <tbody>
                    {cartItems.map((product) => (
                      <CartItem key={product.id} product={product}
onRemove={handleRemoveFromCart} />
                    ))}
                    <tr>
                      <td></td>
                      <td></td>
                      <td><b>Cart Total</b></td>
                      <td colSpan={4}>
                        <b> &#8377;{cartTotal}</b>
                      </td>
                    </tr>
                  </tbody>
                </table>
              }
              <hr />
            </div>
          );
        };
export default Cart;
```

7. **src/components/LoginForm.js**

```
import React from 'react';
import { useDispatch, useSelector } from 'react-redux';
import { useForm } from 'react-hook-form';
import { loginUser } from '../dispatchers/userDispatcher';
import ProductList from './ProductList';

const LoginForm = () => {
  const dispatch = useDispatch();
  const { isLoggedIn } = useSelector((state) => state.user);
  const { register, handleSubmit, formState: { errors } } = useForm();

  const onSubmit = ({ email, password }) => {
    dispatch(loginUser(email, password));
  };

  if (isLoggedIn) {
    return (<ProductList />)
  }
  return (
    <>
      <form onSubmit={handleSubmit(onSubmit)}>
        <h2>Login</h2>
        <div>
          <input type="email" placeholder='Email' {...register('email', {
required: true })} />
          {errors.email && <p>Email is required.</p>}
```

```
                </div>
                <div>
                  <input type="password" placeholder='' {...register('password', {
required: true })} />
                  {errors.password && <p>Password is required.</p>}

                </div>
                <div>
                  <button type="submit">Login</button>
                </div>
              </form>
            </>
          );
};
export default LoginForm;
```

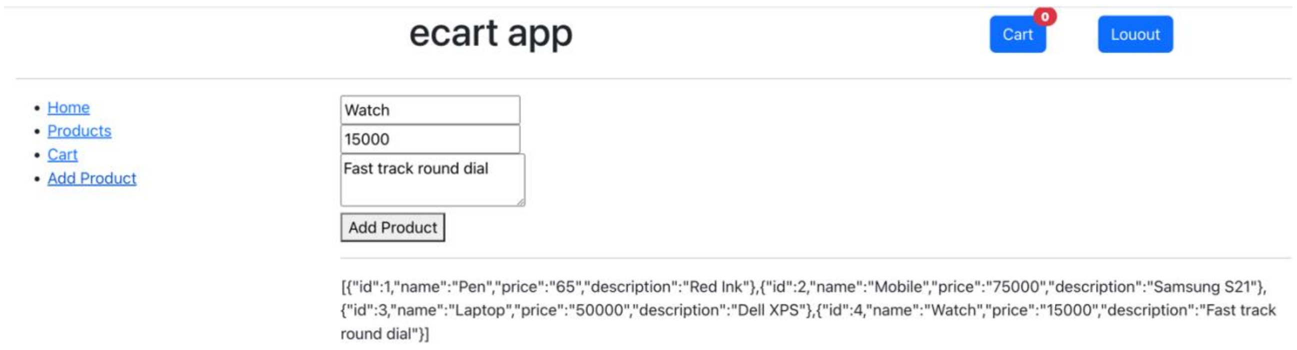
10. Additionally, the **Provider** component should be added to the src/**index.js** file.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
import { Provider } from 'react-redux';
import { configureStore } from '@reduxjs/toolkit';
import rootReducer from './reducers/rootReducer';
import { BrowserRouter } from 'react-router-dom';

const root = ReactDOM.createRoot(document.getElementById('root'));
const store = configureStore({
  reducer: rootReducer
});

root.render(
  <React.StrictMode>
    <BrowserRouter>
    <Provider store={store}>
    <App />
    </Provider>
    </BrowserRouter>
  </React.StrictMode>
);
reportWebVitals();
```

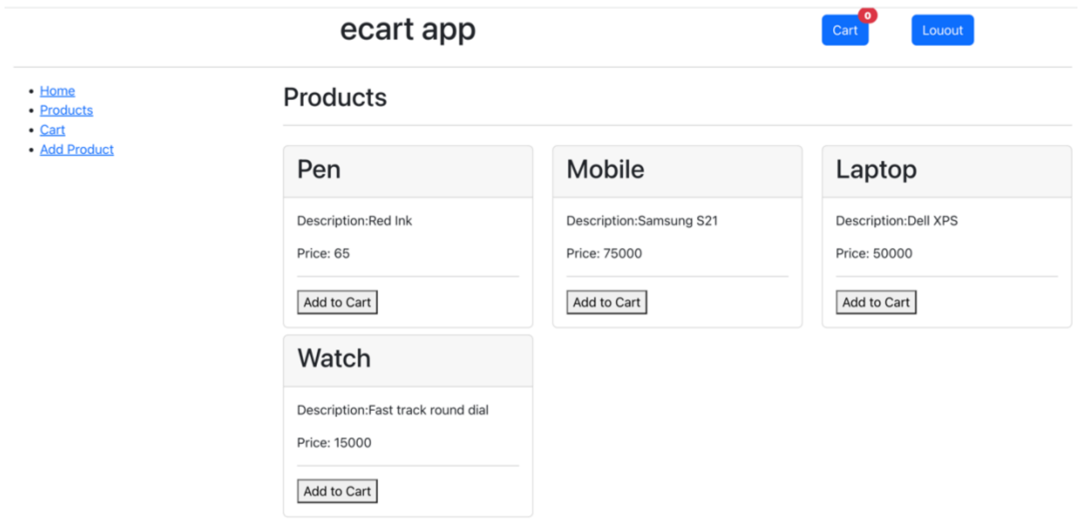
11. Finally add following router code for **src/App.js** component

```
import React from 'react';
import { BrowserRouter as Router, Route, Link, Routes } from 'react-router-dom';
import LoginForm from './components/LoginForm';
import ProductList from './components/ProductList';
import Cart from './components/Cart';
import AddProductForm from './components/AddProductForm';
import 'bootstrap/dist/css/bootstrap.css';
import { useSelector } from 'react-redux';

function App() {
  const cartItems = useSelector(state => state.cart.items);

  return (

    <div className='container'>
      <div className='row'>
        <div className='col-10'>
          <div className='text-center'>
            <h1>ecart app</h1>
          </div>
        </div>
        <div className='col-2 mt-3'>
          <button type="button" className="btn btn-primary position-relative">
            Cart
            <span className="position-absolute top-0 start-100 translate-middle badge
rounded-pill bg-danger">
              <b>{cartItems.length}</b>
              <span className="visually-hidden">unread messages</span>
            </span>
          </button>
        </div>
      </div>
      <hr />
      <div className='row'>
        <div className='col-3'>
          <nav>
            <ul>
              <li>
                <Link to="/">Home</Link>
              </li>
              <li>
                <Link to="products">Products</Link>
              </li>
              <li>
                <Link to="cart">Cart</Link>
              </li>
              <li>
                <Link to="add-product">Add Product</Link>
              </li>
            </ul>
          </nav>
        </div>
        <div className='col-9'>
```

```
          <Routes>
          <Route path="/" element={<ProductList />} />
            <Route path="add-product" element={<AddProductForm />} />
            <Route path="products" element={<ProductList />} />
            <Route path="cart" element={<Cart />} />
          </Routes>
        </div>
      </div>
    </div>
  );
}
export default App;
```



```
[{"id":1,"name":"Pen","price":"65","description":"Red Ink"},{"id":2,"name":"Mobile","price":"75000","description":"Samsung S21"},
{"id":3,"name":"Laptop","price":"50000","description":"Dell XPS"},{"id":4,"name":"Watch","price":"15000","description":"Fast track
round dial"}]
```



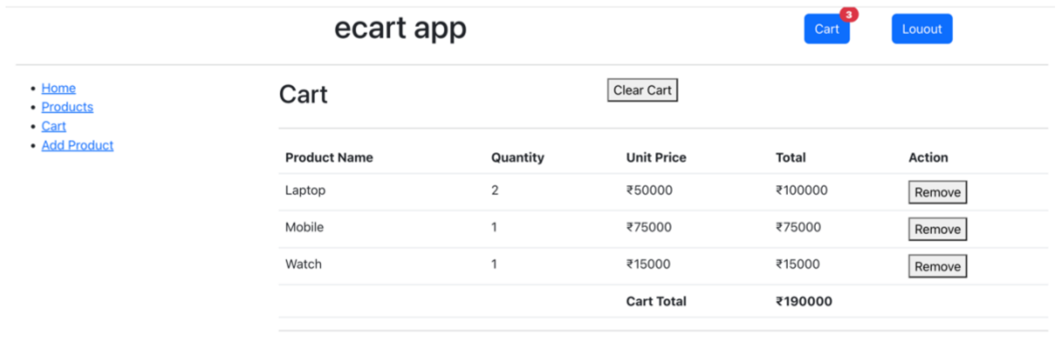
Products

Pen
Description:Red Ink
Price: 65
Add to Cart

Mobile
Description:Samsung S21
Price: 75000
Add to Cart

Laptop
Description:Dell XPS
Price: 50000
Add to Cart

Watch
Description:Fast track round dial
Price: 15000
Add to Cart



Product Name	Quantity	Unit Price	Total	Action
Laptop	2	₹50000	₹100000	Remove
Mobile	1	₹75000	₹75000	Remove
Watch	1	₹15000	₹15000	Remove
		Cart Total	₹190000	

Implementing middleware to redux application

What is middleware?
In React Redux, middleware is a function that intercepts actions dispatched by the application before they reach the reducers, and can perform additional processing. Middleware sits between the dispatch() method and the reducers, allowing for complex asynchronous flows and side effects. A middleware function takes three arguments: store, next, and action. Middleware can be used to add additional functionality to Redux, such as logging, routing, or handling asynchronous requests. To use middleware in a Redux application, you can pass it to the applyMiddleware() method when creating the store.

Middleware in React Redux provides several advantages, including:

1. Reusable code: Middleware allows you to write reusable code that can be used across multiple applications. This can help to reduce the amount of duplicate code in your application and make it easier to maintain.

2. Asynchronous operations: Middleware provides a way to handle asynchronous operations, like fetching data from a server, in a consistent and manageable way. This can help to simplify your application code and make it easier to reason about.

3. Side effects: Middleware can be used to manage side effects in your application, like logging, tracking analytics, or sending messages to external services. By using middleware to manage side effects, you can keep your application code clean and focused on the core functionality.

4. Intercepting actions: Middleware provides a way to intercept and modify actions before they are sent to the reducers. This can be useful for implementing features like authentication, authorization, or error handling.

5. Third-party packages: There are many existing middleware packages available for React Redux that can help to simplify common tasks, like handling asynchronous requests or managing routing.

Overall, middleware provides a way to extend and customize Redux to fit the specific needs of your application. By using middleware, you can add additional functionality to your application, while keeping your code clean and easy to manage.

Types of middleware in react redux

There are many middleware packages available for React Redux. Here are some of the most popular packages:

1. **Redux Thunk:** Redux Thunk is a popular middleware package that allows you to write action creators that return a function instead of an action object. This function can perform asynchronous operations, such as fetching data from an API, before dispatching an action to update the Redux store.
2. **Redux Saga:** Redux Saga is a middleware package that allows you to write long-running background processes, called sagas, that handle asynchronous operations, like fetching data from an API or handling user input. Sagas can communicate with the Redux store and dispatch actions as needed.
3. **Redux Logger:** Redux Logger is a middleware package that can be used to log all the actions that are dispatched in your application. This can be helpful for debugging and understanding the flow of your application.
4. **Redux Promise:** Redux Promise is a middleware package that allows you to write action creators that return a Promise. This can be useful for handling asynchronous operations, like fetching data from an API, in a consistent and manageable way.
5. **Redux Observable:** Redux Observable is a middleware package that allows you to write reactive programming-style code using Observables. This can be useful for handling asynchronous operations and managing side effects in your application.
6. **React Router Redux:** React Router Redux is a middleware package that allows you to manage routing in your application using the React Router library. It can intercept actions that represent changes to the URL and update the store accordingly, which can trigger a change in the UI.
7. **Redux Persist:** Redux Persist is a middleware package that allows you to persist the Redux store to storage, such as local storage or AsyncStorage. This can be useful for preserving the state of your application across sessions or page refreshes.

These are just a few examples of the many middleware packages available for React Redux. By using middleware, you can extend and customize Redux to fit the specific needs of your application, while keeping your code clean and easy to manage.

Redux Thunk

Redux Thunk is a middleware for React Redux that allows you to write action creators that return a function instead of an action object. This function can perform asynchronous operations, such as fetching data from an API, before dispatching an action to update the Redux store.

In Redux, actions are plain objects that describe a change in the application's state. When an action is dispatched, it is immediately sent to the reducers, which update the state accordingly. However, some actions might require asynchronous operations, like waiting for a response from a server. Redux Thunk allows you to dispatch a function instead of an action, which can then perform these asynchronous operations and dispatch the actual action when it is ready.

Suppose you have following code for fetching posts.

```
import axios from 'axios';
function fetchPosts() {
  return axios.get('https://example.com/api/products')
    .then(response => {
      // Transform the data if necessary
      return response.data;
    })
    .catch(error => {
      // Handle any errors
      console.error('Error fetching products:', error);
      throw error;
    });
}
```

Here's an example of a Redux Thunk action creator that fetches data from an API:

```
import { fetchPosts } from './api';
export function loadPosts() {
  return function(dispatch) {
    return fetchPosts().then(posts => {
      dispatch({ type: 'LOAD_POSTS_SUCCESS', payload: posts });
    });
  };
}
```

In this example, the loadPosts() action creator returns a function that takes the dispatch method as an argument. This function then calls the fetchPosts() function, which returns a Promise that resolves to the fetched data. When the Promise is resolved, the function dispatches an action with the fetched data.

To use Redux Thunk in your React Redux application, you need to apply the middleware to your store. Here's an example:

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './reducers';

const store = createStore(rootReducer, applyMiddleware(thunk));
```

In this example, we're using the **applyMiddleware()** method to apply the **thunk** middleware to our store. This allows us to use Redux Thunk action creators in our application.

To install Redux Thunk in your React Redux application, you can use the following command:

```
npm install redux-thunk
```

This command installs the latest version of Redux Thunk from the NPM registry and adds it to your project's node_modules directory.

After installing Redux Thunk, you need to apply it as middleware to your Redux store. Here's an example of how you could do that:

```
import { configureStore } from '@reduxjs/toolkit';
import thunk from 'redux-thunk';
import rootReducer from './reducers';
const store = configureStore({
  reducer: rootReducer,
  middleware: [thunk],
});
```

In this example, we're using the **configureStore** function from **@reduxjs/toolkit** to create our Redux store. We're passing in an object with a reducer key that points to our root reducer, and a middleware key that specifies the middleware to apply to the store. We're using thunk middleware to handle asynchronous actions.

Note that **configureStore** also has many other options that you can use to customize your store, such as enabling dev tools and setting up preloaded state.

Simple application using Redux Thunk

here's an example of how you can use Redux Thunk to handle an asynchronous action in your Redux store:

1. First, install the required packages using **npm** tool
   npm install redux react-redux redux-thunk
2. Create a new file called **store.js** to set up your Redux store:
   ```
   import { configureStore } from '@reduxjs/toolkit';
   import rootReducer from './reducers';
   import thunk from 'redux-thunk';

   const store = configureStore({
     reducer: rootReducer,
     middleware: [thunk],
   });
   export default store;
   ```

In this example, we're using **configureStore** function from **@reduxjs/toolkit** to create the store. We're passing in an object with a **reducer** key that points to our root reducer, and a **middleware** key that specifies the middleware to apply to the store. We're using **thunk** middleware to handle asynchronous actions.

3. Create a new file called **reducers.js** to define your root reducer:

```
import { combineReducers } from 'redux';
import { createSlice } from '@reduxjs/toolkit';

const initialState = {
  loading: false,
  data: null,
  error: null,
};

const slice = createSlice({
  name: 'posts',
  initialState,
  reducers: {
    fetchPostsStart(state) {
      state.loading = true;
      state.data = null;
      state.error = null;
    },
    fetchPostsSuccess(state, action) {
      state.loading = false;
      state.data = action.payload;
      state.error = null;
    },
    fetchPostsError(state, action) {
      state.loading = false;
      state.data = null;
      state.error = action.payload;
    },
  },
});

export const { fetchPostsStart, fetchPostsSuccess, fetchPostsError } = slice.actions;

export default combineReducers({
  posts: slice.reducer,
});
```

In this example, we're using the createSlice function from @reduxjs/toolkit to create a slice of the Redux state to manage our posts. We're defining three actions to handle the different stages of the asynchronous operation: fetchPostsStart, fetchPostsSuccess, and fetchPostsError. We're also combining our slice reducer with other reducers, if any.

4. Create a new file called **actions.js** to define your action creator:

```
import { fetchPostsStart, fetchPostsSuccess, fetchPostsError } from './reducers';

export function fetchPosts() {
  return function(dispatch) {
    dispatch(fetchPostsStart());
    fetch('https://jsonplaceholder.typicode.com/posts')
      .then(response => response.json())
      .then(data => dispatch(fetchPostsSuccess(data)))
      .catch(error => dispatch(fetchPostsError(error.message)));
  };
}
```

In this example, we're defining an **fetchPosts** action creator that returns a function which dispatches three different actions during the asynchronous operation: **fetchPostsStart**, **fetchPostsSuccess**, and **fetchPostsError**. We're using the **fetch** method to make a GET request to an API endpoint that returns a list of posts. We're then parsing the response as JSON data and dispatching **fetchPostsSuccess** with the data, or **fetchPostsError** with the error message.

5. Add following code to index.js

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

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

6. Create a new file called **App.js** to use your store and action creator:

```
import React, { useEffect } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { fetchPosts } from './actions';

function App() {
  const dispatch = useDispatch();
  const posts = useSelector(state => state.posts);

  useEffect(() => {
    dispatch(fetchPosts());
  }, [dispatch]);

  return (
    <div>
      <h1>Posts</h1>
      {posts.loading && <p>Loading...</p>}
      {posts.data && (
        <ul>
          {posts.data.map(post => (
            <li key={post.id}>{post.title}</li>
          ))}
        </ul>
      )}
      {posts.error && <p>Error: {posts.error}</p>}
    </div>
  );
}
export default App;
```