# REDUX

=>HISTORY

* Classic redux and class based components are not that much in use today
* There is a modernized form of redux , REDUX TOOLKIT used nowadays
* Before context and redux , to solve the problem of state management and prop drilling, there was **flux.**
* **FLUX => was created by facebook to solve the problem of state management. It didn’t have a strong data flow , so redux was introduced** (by data flow, it means the mechanism of how data is being send, updated, received **)**
* **Redux is an independent library which can be used with js frameworks for state management.**
* **But redux was also causing some problems like (). So redux toolkit was introduced.**

# File Structure in Redux vs RTK

|  |  |
| --- | --- |
| REDUX  /your-app  |-- /src  |-- /actions  | |-- ActionTypes.js  | |-- counterActions.js  |-- /components  | |-- App.js  |-- /reducers  | |-- index.js  | |-- counterReducer.js  |-- /store  | |-- index.js  |-- index.js | REDUX TOOLKIT  /your-app  |-- /src  |-- /app  | |-- store.js  |-- /features  | |-- counter  | |-- counterSlice.js  |-- /components  | |-- App.js  |-- index.js |

# For Redux

* /actions: Contains all action-related files.
* ActionTypes.js holds the constants for action types.
* counterActions.js contains functions that create actions.
* /components: Holds React components.
* App.js is your main React component that connects to Redux.
* /reducers:
* index.js combines all reducers (using combineReducers if you have more than one).
* counterReducer.js contains the reducer logic for the counter.
* /store:
* index.js sets up the Redux store and integrates middleware if necessary.
* index.js: The entry point of your React application that renders the React component tree.

# For Redux Toolkit

(With Redux Toolkit, the structure can be simplified due to the more integrated approach provided by createSlice, which combines reducers and actions)

* /app:
* store.js sets up the Redux store using configureStore and includes reducers from features.
* /features: Each feature has its own directory, making it scalable for larger projects.
* /counter:
* counterSlice.js includes the reducer and actions for counter features, utilizing createSlice.
* /components:
* App.js is your main React component.
* index.js: The main entry point that renders the React app and integrates the Redux store.

# Comparison and Considerations

# Plain Redux:

* Separates actions and reducers explicitly, which can be clearer when learning Redux or when you need explicit control over each part of the Redux flow.
* More boilerplate, which can become verbose and repetitive in larger applications.

# Redux Toolkit:

* Reduces boilerplate significantly by integrating actions and reducers into single files using slices.
* Encourages better scalability and maintainability practices, especially suitable for larger applications.
* Makes configuration and common Redux setups (like middleware integration) easier and less error-prone.

Conclusion:

Both structures aim to maintain clarity by separating concerns, but Redux Toolkit provides more streamlined, less verbose methods suited for modern application development.

# What are features basically, are they components of app or something else?

When we talk about "features" in Redux Toolkit and modern app setups, we're basically talking about organizing our code based on what different parts of our app do, rather than how they work technically.

So instead of just grouping things like reducers or components together, we group them based on the specific jobs or tasks they handle in our app. **For example, in a social media app, you might have features like posting updates, commenting, or messaging.**

**This approach, called feature-based architecture**, helps us organize our code in a way that makes sense for our users and the functionality they interact with, making it easier to understand and manage as our app grows.

For example, if you were building a shopping cart application, you might have features such as:

* Cart
* Products
* User Authentication
* Checkout Process

Each feature would then contain all the related:

* Redux slices: Combining reducers, actions, and selectors that pertain to a specific domain.
* Components: React components that are used to build that part of the user interface.
* Services: External interactions, such as API calls, would be handled by services or thunks within the feature.
* Tests: Tests that cover the functionalities encapsulated by the feature.

**For instance, if we consider the "Cart" feature in an e-commerce app, the directory might look like this:**

/your-app

|-- /src

|-- /features

|-- /cart

|-- cartSlice.js // Redux logic: actions, reducers, selectors

|-- CartComponent.js // React component specific to the Cart

|-- cartAPI.js // API calls related to cart operations

|-- cartHooks.js // React hooks specific to cart operations

|-- cartStyles.module.css // Styles specific to the cart feature

*(This structure ensures that everything related to the "cart" is located in one place, making development, testing, and debugging much more straightforward.)*

# Does the feature-based architecture in Redux Toolkit eliminate the need for a separate components folder, or should components be integrated within each feature's directory?

You can still have a components folder, but how you organize it in a feature-based architecture can vary based on your project's scale and team preferences.

### 1. Local Components within Features

In a true feature-based structure, each feature folder might contain its own set of components that are specific to that feature. For example:

/your-app

|-- /src

|-- /features

|-- /cart

|-- components

|-- CartButton.js

|-- CartItemList.js

|-- cartSlice.js

|-- cartAPI.js

|-- /products

|-- components

|-- ProductList.js

|-- ProductItem.js

|-- productsSlice.js

|-- productsAPI.js

### 2. Shared Global Components

For components that are reused across different features, you might still maintain a global components directory. This would house reusable UI elements like buttons, modals, layout components, etc., which are not specific to any one feature. For example:

/your-app

|-- /src

|-- /components

|-- Button.js

|-- Modal.js

|-- Layout.js

|-- /features

|-- /cart

|-- CartComponent.js

|-- cartSlice.js

|-- /products

|-- ProductList.js

|-- productsSlice.js

# What is Redux?

Redux is a state management library for JavaScript applications. It helps you manage the state of your app in a predictable way by storing the entire state in a single place called the store. Components can dispatch actions to update the state, and these updates are handled by pure functions called reducers. Redux makes it easier to debug, test, and maintain your application, especially as it grows larger.

## Core Concepts:

**Store:** This is where your application's state lives. It's like a big JavaScript object that holds all your data.

**Actions:** These are payloads of information that send data from your application to your store. They are plain JavaScript objects.

**Reducers:** Reducers specify how the application's state changes in response to actions sent to the store. They are pure functions (functions that return the same output for the same input).

**Dispatch:** This is the way you send actions to the store to update the state.

**Selectors:** These are functions that extract specific pieces of data from the store state.

## How Does Redux Work?

1. Your application dispatches an action.
2. The Redux store receives the action.
3. Based on the action type, the store calls the appropriate reducer.
4. The reducer takes the current state and the action, computes the new state, and returns it.
5. The store updates its state to the new state returned by the reducer.
6. Any UI components subscribed to the store will receive the updated state and re-render.

## Setting Up Redux:

1. Install Redux and React-Redux packages in your project.
2. Create a Redux store using createStore() function from Redux.
3. Write reducers to specify how the state should change in response to actions.
4. Connect your React components to the Redux store using connect() from React-Redux.
5. Dispatch actions from your components to update the state.

## Example:

Let's say you have a counter app. You dispatch an action like { type: 'INCREMENT' }. Your reducer listens for this action type and increments the counter in the state. Then your UI re-renders with the updated counter.

## Counter example using Redux without Redux Toolkit:

**// ActionTypes.js**

export const INCREMENT = 'INCREMENT';

export const DECREMENT = 'DECREMENT';

**// actions.js**

import { INCREMENT, DECREMENT } from './ActionTypes';

export const increment = () => ({

type: INCREMENT

});

export const decrement = () => ({

type: DECREMENT

});

**// reducers.js**

import { INCREMENT, DECREMENT } from './ActionTypes';

const initialState = {

value: 0

};

const counterReducer = (state = initialState, action) => {

switch (action.type) {

case INCREMENT:

return {

...state,

value: state.value + 1

};

case DECREMENT:

return {

...state,

value: state.value - 1

};

default:

return state;

}

};

export default counterReducer;

**// store.js**

import { createStore } from 'redux';

import counterReducer from './reducers';

const store = createStore(counterReducer);

export default store;

**// App.js**

import React from 'react';

import { connect } from 'react-redux';

import { increment, decrement } from './actions';

const App = ({ value, increment, decrement }) => {

return (

<div>

<h1>Counter: {value}</h1>

<button onClick={increment}>Increment</button>

<button onClick={decrement}>Decrement</button>

</div>

);

};

const mapStateToProps = (state) => ({

value: state.value

});

const mapDispatchToProps = {

increment,

decrement

};

export default connect(mapStateToProps, mapDispatchToProps)(App);

In this example:

* We define action types in a separate file ActionTypes.js.
* Action creators are defined in actions.js.
* Reducers are defined in reducers.js.
* We create the Redux store using createStore from Redux in store.js.
* Finally, in App.js, we connect our React component to Redux using connect and define mapStateToProps and mapDispatchToProps functions to map the Redux state and dispatch actions to props.

## As you can see, Redux alone requires more boilerplate code compared to Redux Toolkit. Redux Toolkit simplifies this process by providing utilities like createSlice and configureStore, which automatically generate action creators, action types, and handle store configuration.

# Redux Toolkit (RTK)

Redux Toolkit is an official, recommended library for simplifying Redux usage in JavaScript applications. It reduces boilerplate code and includes best practices by default. Redux Toolkit provides utilities like configureStore for easy store setup, createSlice for generating reducers and actions, and createAsyncThunk for handling asynchronous logic. This makes managing application state more efficient and less error-prone.

## Why Redux Toolkit (RTK) over Redux?

Redux Toolkit (RTK) is preferred over plain Redux because it simplifies the process of state management by reducing boilerplate code and incorporating best practices by default. RTK provides tools like configureStore for easy store setup, createSlice for combining actions and reducers, and createAsyncThunk for handling asynchronous logic, making it more efficient and less error-prone to use Redux in applications.

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

const counterSlice = createSlice({

name: 'counter',

initialState: {

value: 0

},

reducers: {

incremented(state) {

// `state` is a draft state produced by Immer, so you can write mutations directly.

state.value += 1;

},

decremented(state) {

state.value -= 1;

}

}

});

const store = configureStore({

reducer: {

counter: counterSlice.reducer

}

});

// Action creators are generated for each case reducer function

const { incremented, decremented } = counterSlice.actions;

// Now you can dispatch these actions

store.dispatch(incremented());

store.dispatch(decremented());

## Data Flow In Redux Toolkit (RTK)

Component (User interacts) → Dispatch Action → Reducer (Updates state) →

Store (State changed) → Component (Re-renders with new state)

1. **Action Dispatch:**

The process starts when an action is dispatched. An action is just a plain JavaScript object that describes what happened. In Redux Toolkit, you often use createAction or actions generated by createSlice to define these actions.

1. **Reducer Function:**

Actions are sent to a reducer function. Reducers are pure functions that take the current state and an action, and return a new state based on that action. In Redux Toolkit, you use createReducer or createSlice to define reducers. The reducer function will check which action has been performed and will modify the state accordingly.

1. **Store Updates:**

The store is the central repository for all the state in your application. After the reducer produces a new state, the store is updated with this new state.

1. **UI Reacts:**

The UI, usually a React component, listens for changes in the store. This is typically done using the useSelector hook from react-redux, which allows components to subscribe to the store. When the state changes, the components that depend on that piece of the state will re-render with the new data.

1. **State is Passed as Props:**

State from the store is passed down to components as props, allowing them to re-render with updated data. This is how the application’s UI stays in sync with the state.

1. **Components Dispatch Actions:**

Components can also dispatch actions based on user interactions, such as clicking a button. This action is then processed in the same flow, modifying the state and updating the UI correspondingly.

## Basic steps to use Redux Toolkit (RTK) in a project :

# 1. Install Redux Toolkit and React-Redux:

*npm install @reduxjs/toolkit react-redux*

# 2. Set Up the Store:

* *Create a store.js file.*
* *Use configureStore to create the Redux store.*

// src/store.js

import { configureStore } from '@reduxjs/toolkit';

import rootReducer from './reducers'; // Assuming you have a root reducer

const store = configureStore({

reducer: rootReducer,

});

export default store;

# 3. Create a Slice:

*Use createSlice to define a slice of the state, including actions and reducers.*

// src/features/counter/counterSlice.js

import { createSlice } from '@reduxjs/toolkit';

const counterSlice = createSlice({

name: 'counter',

initialState: { value: 0 },

reducers: {

increment: state => { state.value += 1 },

decrement: state => { state.value -= 1 }

}

});

export const { increment, decrement } = counterSlice.actions;

export default counterSlice.reducer;

# 4. Integrate Slice Reducer into the Store:

*Add the slice reducer to your store configuration.*

// src/store.js

import { configureStore } from '@reduxjs/toolkit';

import counterReducer from './features/counter/counterSlice';

const store = configureStore({

reducer: {

counter: counterReducer,

},

});

export default store;

# 5. Provide the Store to Your App:

*Wrap your application with the Provider component from react-redux and pass the store.*

// src/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')

);

# 6. Use Redux State and Actions in Components:

*Use useSelector to access state and useDispatch to dispatch actions.*

// src/App.js

import React from 'react';

import { useSelector, useDispatch } from 'react-redux';

import { increment, decrement } from './features/counter/counterSlice';

function App() {

const count = useSelector(state => state.counter.value);

const dispatch = useDispatch();

return (

<div>

<h1>{count}</h1>

<button onClick={() => dispatch(increment())}>Increment</button>

<button onClick={() => dispatch(decrement())}>Decrement</button>

</div>

);

}

export default App;