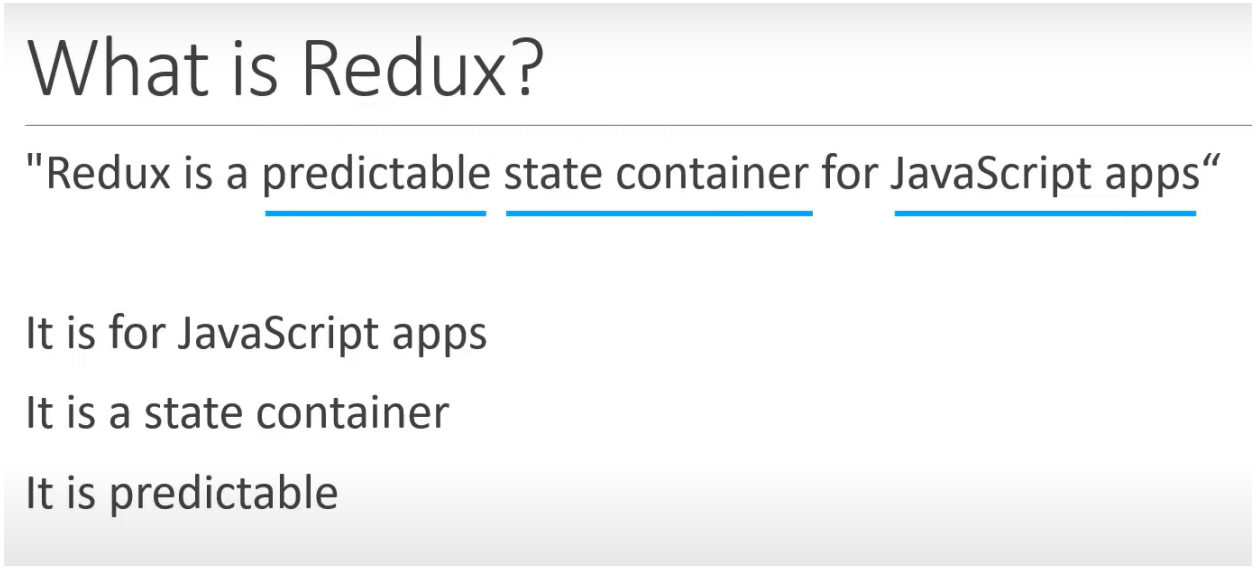
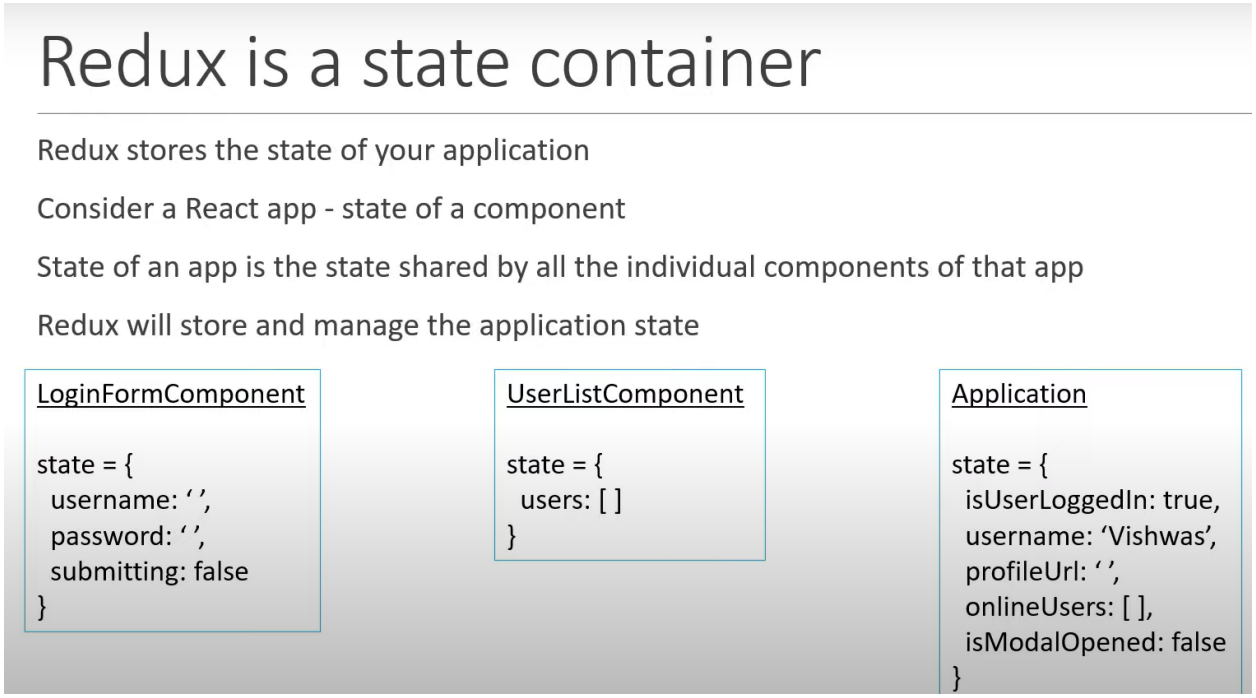
Redux





Redux is a predictable state container for JavaScript applications. This means that **you store all of your application state in one place and can know what the state is at any given point in time**.

Why Redux?

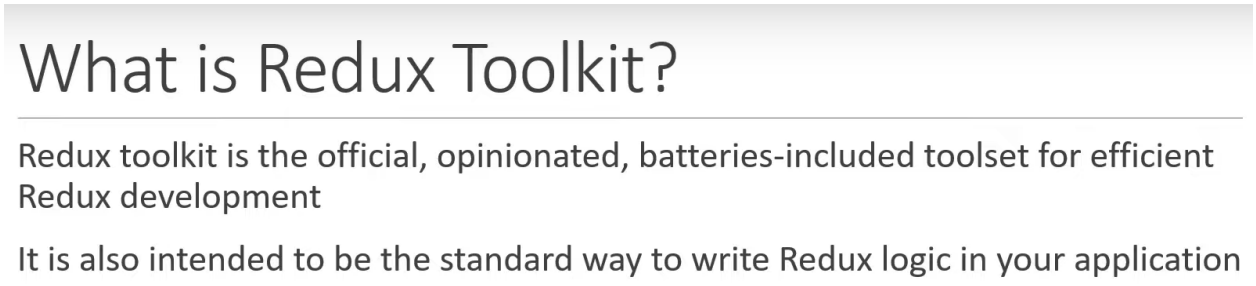
If you want to manage the global state of your application in a predictable way, redux can help you

The patterns and tools provided by Redux make it easier to understand when, where, why, and how the state in your application is being updated, and how your application logic will behave when those changes occur

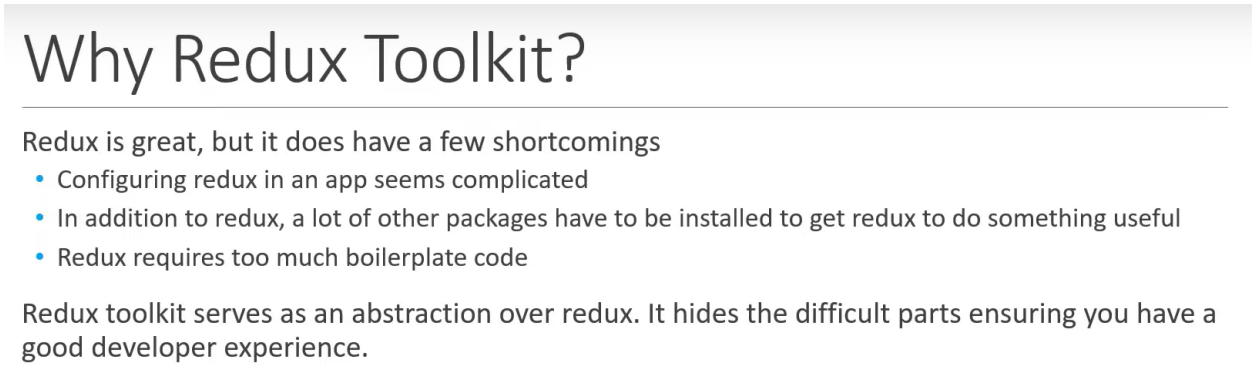
Redux guides you towards writing code that is predictable and testable, which helps give you confidence that your application will work as expected

Redux ToolKit

Redux Toolkit is **a set of tools that helps simplify Redux development**. It includes utilities for creating and managing Redux stores, as well as for writing Redux actions and reducers.



Why Redux ToolKit over Redux?

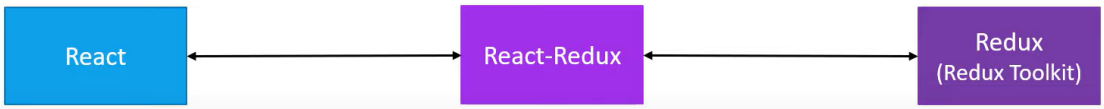


Redux Toolkit does not need a UI Library to work with. (Example: React, Vue etc). It can work with plain JS as well.

React Redux Package

React-Redux

React-Redux is the official Redux UI binding library for React



Summary

React is a library used to build user interfaces

Redux is a library for managing state in a predictable way in JavaScript applications

Redux toolkit is a library for efficient redux development

React-redux is a library that provides bindings to use React and Redux (Toolkit) together in an application

Few points before we proceed

The most basic mistake you can do is learning redux and react in parallel

"When should I use redux in my react application?"

- You have large amounts of application state that are needed in many places in the app
- The app state is updated frequently over time
- The logic to update that state may be complex
- The app has a medium or large-sized codebase, and might be worked on by many people

Redux has a learning curve

Course Structure

1. We will do Redux first.
2. Then Redux Toolkit.
3. Then React Redux Package,

Part-1 - Redux

Setting Up.

1. Open VS Code in any Folder
2. npm init –yes
3. npm install redux

Action, Reducer, Store - Core Concepts

Three Core Concepts

Cake Shop

Entities

Shop – Stores cakes on a shelf
Shopkeeper – Behind the counter
Customer – At the store entrance

Activities

Customer – Order a cake
Shopkeeper – Box a cake from the shelf
– Receipt to keep track

Three Core Concepts contd.

Cake Shop Scenario	Redux	Purpose
Shop	Store	Holds the state of your application
Cake ordered	Action	Describes what happened
Shopkeeper	Reducer	Ties the store and actions together

A **store** that holds the state of your application.

An **action** that describes what happened in the application.

A **reducer** which handles the action and decides how to update the state.

Three Principles

Principle 1:

First Principle

"**The global state of your application is stored as an object inside a single store"**

Maintain our application state in a single object which would be managed by the Redux store

Cake Shop –

Let's assume we are tracking the number of cakes on the shelf

```
{
  numberOfCakes: 10
}
```

Principle 2:

Second Principle

"The only way to change the state is to dispatch an action, **an object that describes what happened"**

To update the state of your app, you need to let Redux know about that with an action

Not allowed to directly update the state object

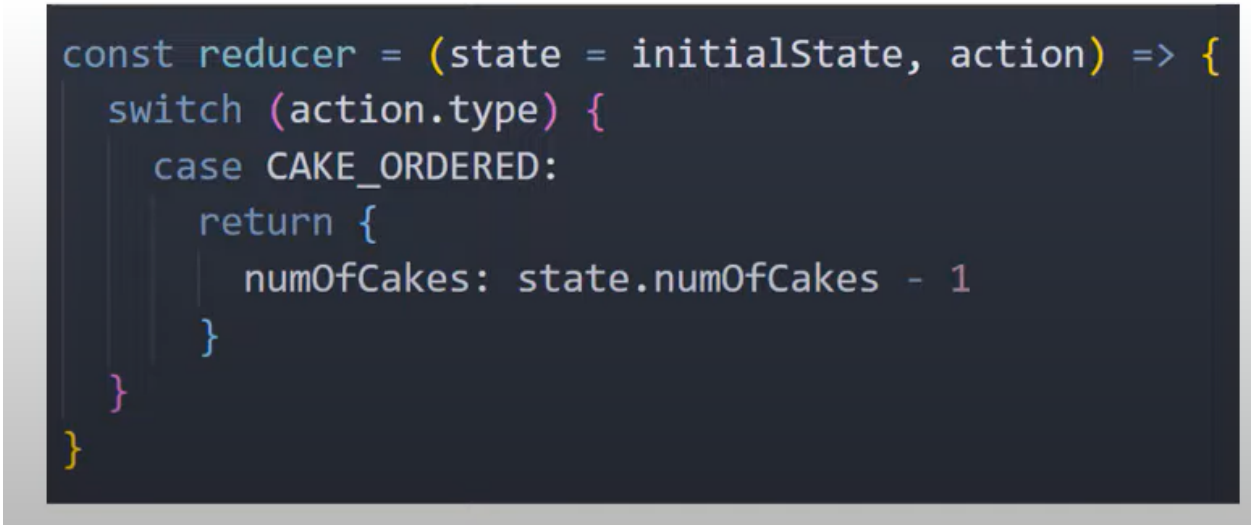
Principle 3:

Third Principle

"To specify how the state tree is updated based on actions, you write pure reducers"

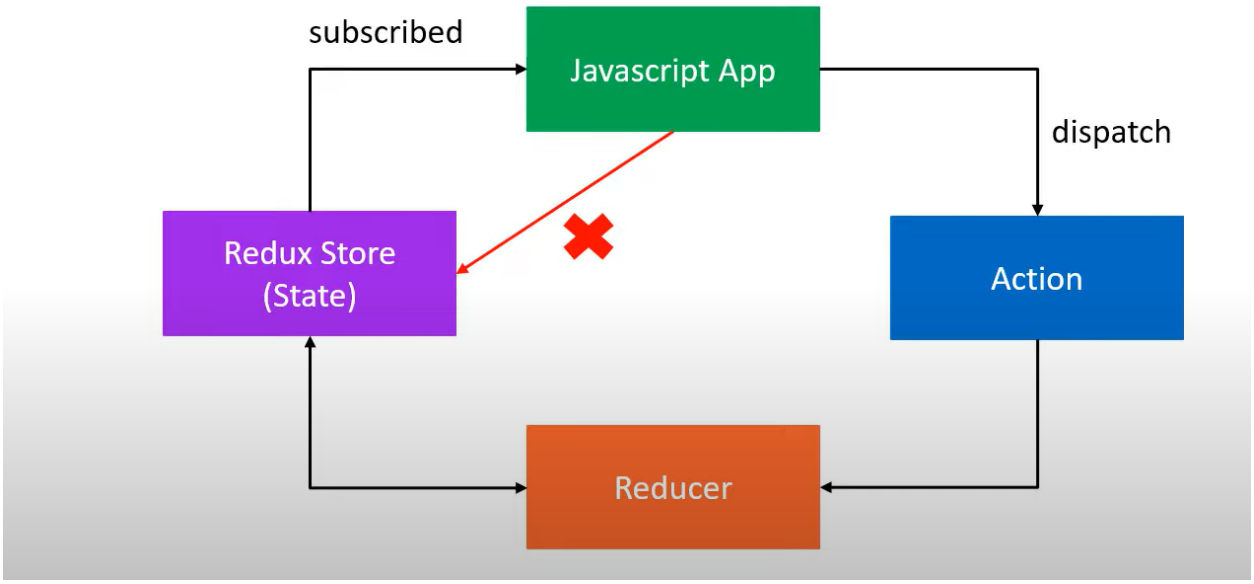
Reducer - (previousState, action) => newState

Pure Reducers are the pure functions that take previous state and action as an input and returns a new state.
Being a pure function, instead of updating the previous state, it should return a new state.



```
const reducer = (state = initialState, action) => {
  switch (action.type) {
    case CAKE_ORDERED:
      return {
        numOfCakes: state.numOfCakes - 1
      }
  }
}
```

Three Principles Overview



Action

Actions

The only way your application can interact with the store

Carry some information from your app to the redux store

Plain JavaScript objects

Have a 'type' property that describes something that happened in the application.

The 'type' property is typically defined as string constants

Two things to remember:
An Action is an object with "type" property.
Action Creator is a function that returns an action object.

```
function actionCakeOrder() {
    return {
        type: "Cake Order",
        quantity: 1
    }
}
```

Action Creator Function

- We used action creator function instead of passing object directly because we might dispatch an action in several files. After a few days we realize that we need to add one more property in an action object.

- Action Creator being a common function used at all the places in our app will solve this problem rather than updating action object manually at all the locations.

Reducers

Reducers

Specify how the app's state changes in response to actions sent to the store

Function that accepts state and action as arguments, and returns the next state of the application

(previousState, action) => newState

```
const initialState = {
    numOfCakes: 1
}


const reducer = (state = initialState, action) => {
    switch (action.type) {
        case "CAKE_ORDER":
            return {
                ...state,
                numOfCakes: state.numOfCakes - 1
            }
        default:
            return state
    }
}
```

Store

Redux Store

One store for the entire application

Responsibilities –

- Holds application state
- Allows access to state via ***getState()***
- Allows state to be updated via ***dispatch(action)***
- Registers listeners via ***subscribe(listener)***
- Handles unregistering of listeners via the function returned by **subscribe(listener)**

Creating a Store

Import redux package

For React:
import redux from “redux”

For NodeJS App:
const redux = require(“redux”)

1. Creating a store

```
const store = redux.createStore(reducer);
```

While creating a store, we need to pass a reducer function.

2. Accessing State:

```
console.log("INITIAL STATE:", store.getState())
```

3. Subscribing to store:

```
store.subscribe(() => {
    console.log("UPDATED STATE:", store.getState())
})
```

We can add listeners which will be triggered when the store is updated.

4. Dispatching an action:

```
store.dispatch(actionCakeOrder());
```

actionCakeOrder() returns an object with “type” property.

5. Unsubscribing to store

```
const unsubsribe = store.subscribe(() => {
    console.log("UPDATED STATE:", store.getState())
})
```

store.**subscribe**(**…**) returns unsubscribe function.

Example:

```
// Creating Store
const store = redux.createStore(reducer);

console.log("INITIAL STATE:", store.getState())

const unsubsribe = store.subscribe(() => {
    console.log("UPDATED STATE:", store.getState())
})

store.dispatch(actionCakeOrder());
store.dispatch(actionCakeOrder());

unsubsribe();

store.dispatch(actionCakeOrder());

console.log("LAST STATE:", store.getState())
```

Output:

```
INITIAL STATE: { numOfCakes: 10 }
UPDATED STATE: { numOfCakes: 9 }
UPDATED STATE: { numOfCakes: 8 }
LAST STATE: { numOfCakes: 7 }
```

Binding Action Creators

Alternate of calling Actions. It binds all the action creators in a single object and now this can be shared across different files.
To be honest: Not much useful these days.

```
// Creating Store
const store = redux.createStore(reducer);

const actions = redux.bindActionCreators(
    {
        actionCakeOrder,
        actionAddNewCake
    },
    store.dispatch
)
```

```
actions.actionCakeOrder();
actions.actionCakeOrder();
actions.actionAddNewCake(10);
```

Multiple Reducers

Considering a Cake Shop example, now the shopkeeper wants to sell Ice Cream also.
But managing two things at a time will be hectic for him.

In Future, More items can also come.
What If, there is one individual shopkeeper for each item.

Shopkeeper One => For Selling Cakes
Shopkeeper Two => For Selling IceCreams

Now the work will be properly distributed and managed.

Aligning this example with Redux, Shopkeeper is acting as a reducer.

For Selling Cakes => We can have Cake Reducer
For Selling IceCreams => We can have IceCream Reducer

For Selling a Cake to Customer, Shopkeeper is acting as a reducer.

We can have multiple reducers in a single store, which will help us to manage our code easily.

We can combine reducers using **redux.combineReducers({...})**

```
// Cake Reducer
const cakeReducer = (state = cakeInitialState, action) => {···
}

// Icecream Reducer
const icecreamReducer = (state = icecreamInitialState, action) => {···
}
```

```
// Root Reducer
const rootReducer = redux.combineReducers({
    cakes: cakeReducer,
    icecreams: icecreamReducer
})


// Creating Store
const store = redux.createStore(rootReducer);
```

```
store.dispatch(orderCake(5));
```

Immer

Whenever we update any state, we do not directly mutate or update the object. We create a copy of the existing state and then update required property in the copied object.

There can be cases where the object is very large and is at several nested levels.

Example:
Now we want to update “state” property.

```
const initialState = {
    name: "Param",
    address: {
        city: "Phagwara",
        state: "Punjab"
    }
}
```

OLD WAY

```
        case "UPDATE":
            //OLD WAY
            return {
                ...state,
                address: {
                    ...state.address,
                    state: "Karnataka"
                }
            }
```

With Immer

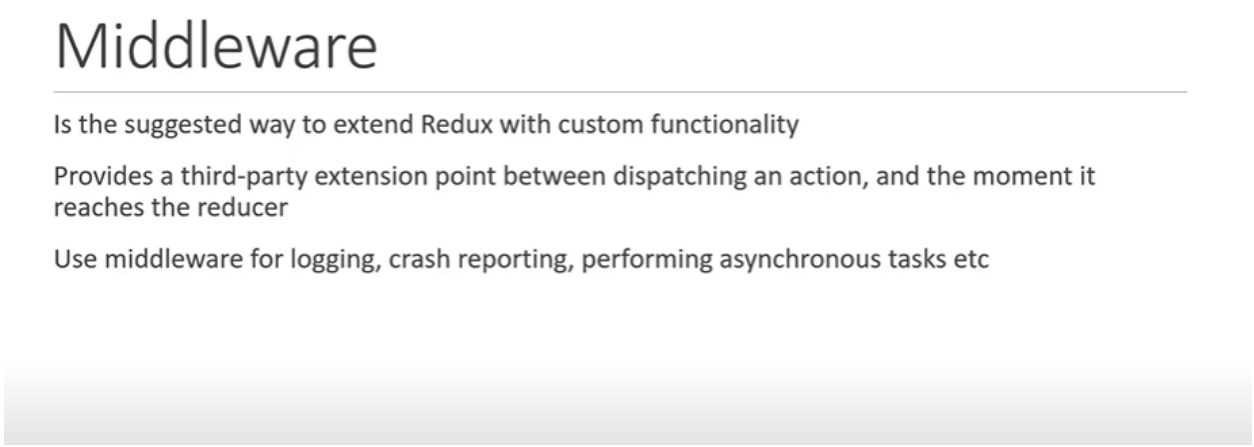
```
const { produce } = require("immer");
```

produce(param1, param2 **)**
param1 => Original State Object
param2 => Arrow function containing Copy of State Object which we can mutate directly.

produce will automatically return the required object.

```
// WITH IMMER
return produce(state, (draft) => {
    draft.address.state = "Karnataka"
})
```

Middleware



To apply Middlewares, Redux library provides us function:
applyMiddleware(yourMiddleware)

For Example:
“redux-logger” is the middleware which we are using in the below example.
npm install redux-logger

```
const redux = require("redux");
const reduxLogger = require("redux-logger")

// Destructuring
const { applyMiddleware } = redux;
const { createLogger } = reduxLogger;
```

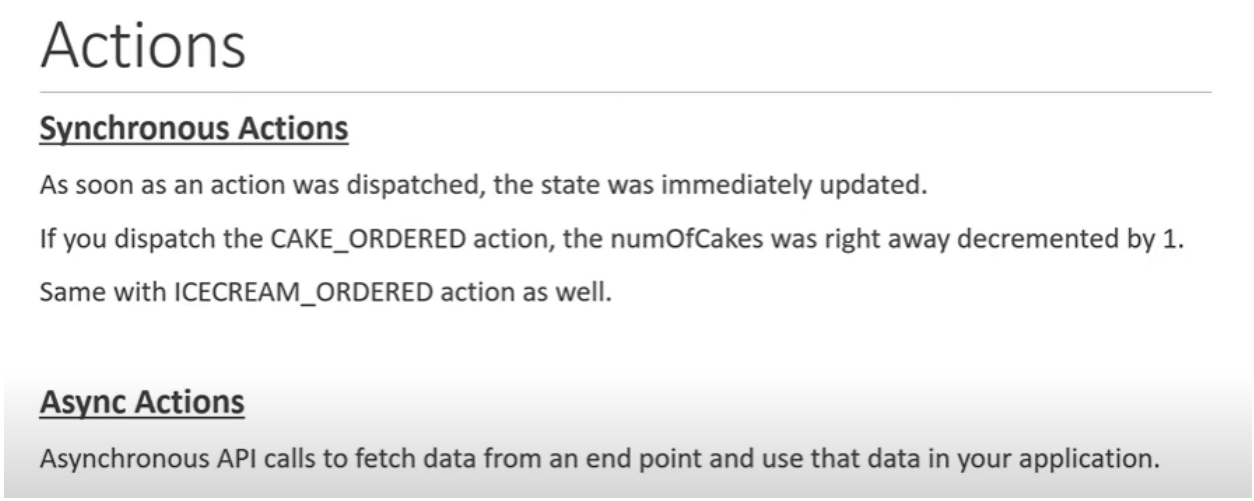
```
// Creating Logger Middleware
const loggerMiddleware = createLogger();

// Create Store & Passing Middleware
const store = redux.createStore(cakeReducer, applyMiddleware(loggerMiddleware));
```

```
store.dispatch(orderCake(3));
store.dispatch(orderCake(2));
store.dispatch(restockCake(5));
```

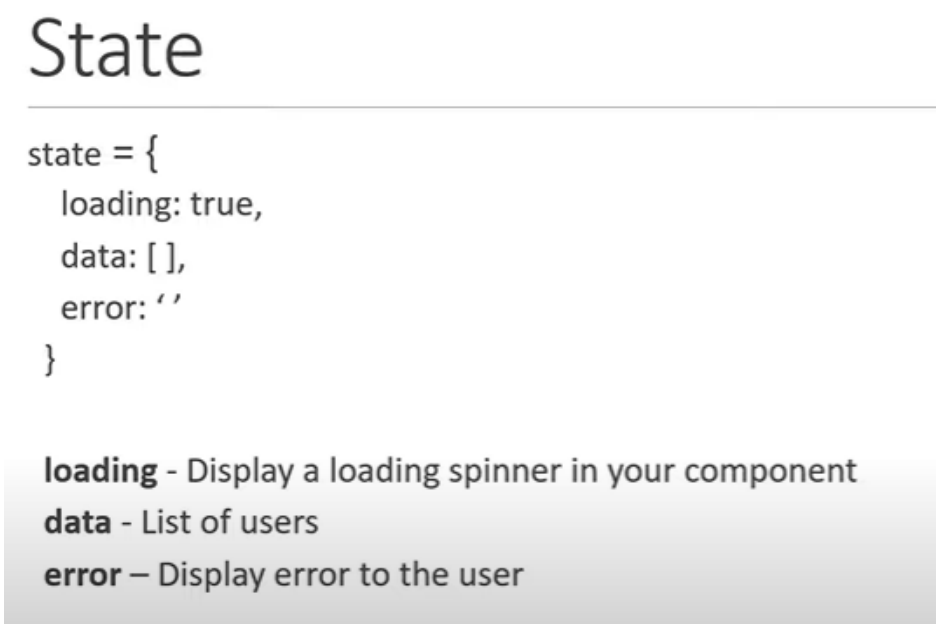
Output: **node index**

```
PS D:\Learning\Redux Toolkit Course\Redux>
 action ORDER_CAKE @ 11:28:42.932
   prev state { numOfCakes: 10 }
   action     { type: 'ORDER_CAKE', payload: 3 }
   next state { numOfCakes: 7 }
 action ORDER_CAKE @ 11:28:42.939
   prev state { numOfCakes: 7 }
   action     { type: 'ORDER_CAKE', payload: 2 }
   next state { numOfCakes: 5 }
 action RESTOCK_CAKE @ 11:28:42.941
   prev state { numOfCakes: 5 }
   action     { type: 'RESTOCK_CAKE', payload: 5 }
   next state { numOfCakes: 10 }
```

Async Actions



Now we will perform Async Actions in our App. But before that, we will maintain:



Actions

FETCH_USERS_REQUESTED – Fetch list of users

FETCH_USERS_SUCCEEDED – Fetched successfully

FETCH_USERS_FAILED – Error when fetching the data

Reducers

case: **FETCH_USERS_REQUESTED**
loading: true

case: **FETCH_USERS_SUCCEEDED**
loading: false
users: data (from API)

case: **FETCH_USERS_FAILED**
loading: false
error: error (from API)

Async action creators

axios

Requests to an API end point

redux-thunk

Define async action creators

Middleware

Redux Thunk

What we learnt so far is that:
Action Creator is a function that returns an object which has the property “type”.

But “redux-thunk” brings new thing to the table. It has the ability to consume Action Creators that returns function instead of an object.

```
const axios = require("axios");
const redux = require("redux");
const thunkMiddleware = require("redux-thunk").default;

const { applyMiddleware } = redux;

// Initial State
const initialState = {
    loading: false,
    users: [],
    error: ""
}

// Actions
const FETCH_USERS_REQUESTED = "FETCH_USERS_REQUESTED";
const FETCH_USERS_SUCCEEDED = "FETCH_USERS_SUCCEEDED";
const FETCH_USERS_FAILED = "FETCH_USERS_FAILED";


// Action Creators
const fetchUsersRequest = () => {···
}

const fetchUsersSuccess = (users) => {···
}

const fetchUsersError = (error) => {···
}
```

```
// Reducer
const reducer = (state = initialState, action) => {···
}

// Async Action Creator
const fetchUsers = () => {
    return function (dispatch) {
        dispatch(fetchUsersRequest);

        axios.get("https://jsonplaceholder.typicode.com/posts")
            .then((response) => {
                const users = response.data.map(user => user.id)
                dispatch(fetchUsersSuccess(users))
            })
            .catch((error) => {
                dispatch(fetchUsersError(error.message))
            })
    }
}

const store = redux.createStore(reducer, applyMiddleware(thunkMiddleware));
store.subscribe(() => {
    console.log(store.getState())
})
store.dispatch(fetchUsers());
```

OUTPUT:

```
PS D:\Learning\Redux Toolkit Course\Redux> node index
{
  loading: false,
  users: [
     1,  2,  3,   4,  5,  6,  7,  8,  9, 10, 11, 12,
    13, 14, 15,  16, 17, 18, 19, 20, 21, 22, 23, 24,
    25, 26, 27,  28, 29, 30, 31, 32, 33, 34, 35, 36,
    37, 38, 39,  40, 41, 42, 43, 44, 45, 46, 47, 48,
    49, 50, 51,  52, 53, 54, 55, 56, 57, 58, 59, 60,
    61, 62, 63,  64, 65, 66, 67, 68, 69, 70, 71, 72,
    73, 74, 75,  76, 77, 78, 79, 80, 81, 82, 83, 84,
    85, 86, 87,  88, 89, 90, 91, 92, 93, 94, 95, 96,
    97, 98, 99, 100
  ],
  error: ''
}
```

End of Redux

Redux concerns

Redux requires too much boilerplate code

- Action
- Action object
- Action creator
- Switch statement in a reducer

A lot of other packages have to be installed to work with redux

- Redux-thunk
- Immer
- Redux-devtools

There was a need to improve the developer experience for redux

Part-2 - Redux Toolkit

Redux Toolkit

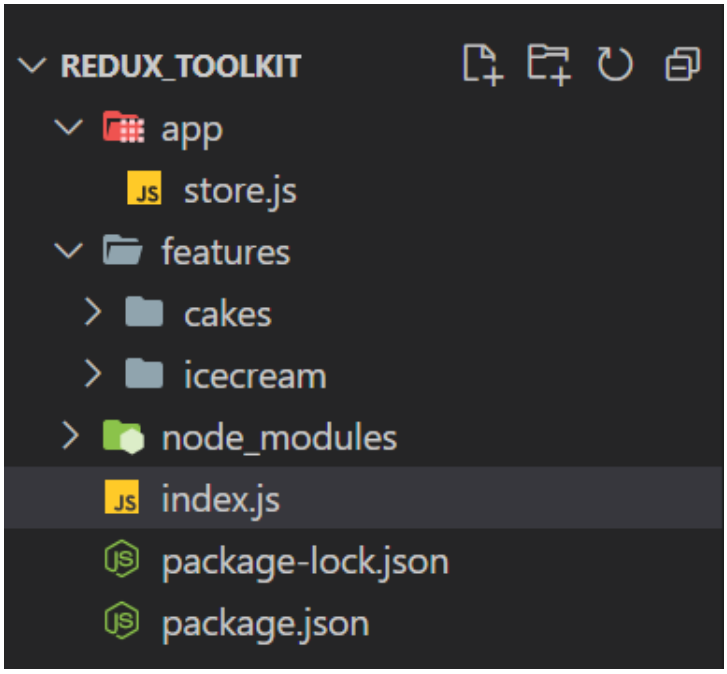
Redux toolkit is the official, opinionated, batteries-included toolset for efficient Redux development

- Abstract over the setup process
- Handle the most common use cases
- Include some useful utilities

npm init –yes

```
npm i @reduxjs/toolkit
```

We will be following Redux Toolkit opinionated Folder Structure for our Cakes & IceCream example.



- As per Redux Toolkit, Recommendation is to group together the reducer logic and actions for a single feature in a single file.
- FileName should contain a “**slice”** as a suffix. (Eg: cake**Slice**.js)

Creating Slice

A "slice" is a collection of Redux reducer logic and actions for a single feature in your app, typically defined together in a single file.

```
const reduxToolkit = require("@reduxjs/toolkit");
const { createSlice } = reduxToolkit;

// Initial State
const initialState = {
    numOfCakes: 10
}

// Slice
const cakeSlice = createSlice({
    name: "cake",
    initialState,
    reducers: {
        // Actions
        "ordered": (state, action) => {
            state.numOfCakes -= action.payload
        },
        "restock": (state, action) => {
            state.numOfCakes += action.payload
        },
    }
})

// Default Export
module.exports = cakeSlice.reducer;

// Named Export
module.exports.cakeActions = cakeSlice.actions;
```

- **createSlice({ })** takes one object as an argument.
- Object must have three properties:
    a. **“name”** of the slice.
    b. **“initialState”** of slice.
    c. **“reducers”** object containing actions as “key” name and callback function as its “value”.
- Callback function has (state, action) as parameters.
- Inside the callback function, we can now mutate the state directly. Under the hood, Immer will take care of this by default.
- Also, we do not need to explicitly return a new state from the callback function.

After creating a slice, We need to export the **reducer** and **actions** of the slice.

1. **Exporting Reducer** and importing it in store, so that it can be linked in the store.
2. **Exporting Actions** and importing where required, so that they can be dispatched to update the store.

Creating Store

In Redux Toolkit, the store is created by the configureStore({ }) function.

app > store.js > ...

```
1   const reduxToolkit = require("@reduxjs/toolkit");
2   const cakeReducer = require("../features/cakes/cakeSlice");
3   const { configureStore } = reduxToolkit;
4
5   const store = configureStore({
6       reducer: {
7           cake: cakeReducer
8       }
9   })
10
11  module.exports = store;
```

Using Store and Dispatching Actions

```
const store = require("./app/store");
const { cakeActions } = require("./features/cakes/cakeSlice")

store.subscribe(() => {
    console.log(store.getState())
})

store.dispatch(cakeActions.ordered(2));
store.dispatch(cakeActions.ordered(3));
store.dispatch(cakeActions.restock(10));
```

Adding Middleware

```
const reduxToolkit = require("@reduxjs/toolkit");
const cakeReducer = require("../features/cakes/cakeSlice");
const icecreamReducer = require("../features/icecream/icecreamSlice");
const reduxLogger = require("redux-logger");

const { configureStore } = reduxToolkit;
const { createLogger } = reduxLogger;

const logger = createLogger();

const store = configureStore({
    reducer: {
        cake: cakeReducer,
        icecream: icecreamReducer
    },
    middleware: (getDefaultMiddleware) => getDefaultMiddleware().concat(logger)
})

module.exports = store;
```

Extra Reducers

- Consider use-case: On Each Order of Cake, We are giving one ice cream free with it.
- Means If we order two cakes, two ice-creams will also be given to that customer.

- In Redux Toolkit, So far we have learnt that **cakeSlice.js** can handle state management related to cakes only.
- Same **icecreamSlice.js** will handle state related to iceCreams only.

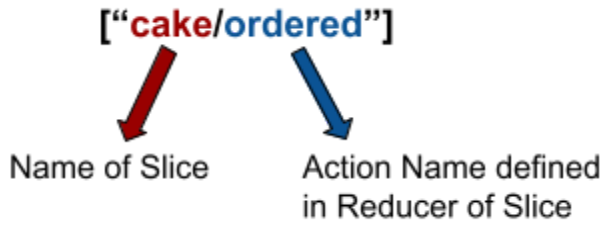
- But as per our use case, We want to update icecream state as well on each order of cake.
- Redux Toolkit provides a way to do so.

- If we want **icecreamSlice.js** to respond to the other action types which are defined in **cakeSlice.js,** then this is doable by adding **extraReducers: { }** property.

```
// Slice
const icecreamSlice = createSlice({
    name: "icecream",
    initialState,
    reducers: {
        // Actions
        "ordered": (state, action) => {
            state.numOfIcecreams -= action.payload
        },
        "restock": (state, action) => {
            state.numOfIcecreams += action.payload
        },
    },
    extraReducers: {
        ["cake/ordered"]: (state, action) => {
            state.numOfIcecreams -= action.payload
        },
    }
})
```

In above example:
In **icecreamSlice.js** we added **extraReducers** property.
In extraReducers property, we attached a callback with **“ordered”** action which was defined in **cakeSlice.js**

This means when **“ordered”** action is dispatched from cakeSlice.js, callback attached to **[“cake/ordered”]** in icrecreamSlice.js will also be triggered, which will update icecream state.



Syntax for adding actions in extraReducers:

```
extraReducers: {
    ["cake/ordered"]: (state, action) => {
        state.numOfIcecreams -= action.payload
    },
}
```

Recommended Approach: Using Build function

```
extraReducers:(builder) => {
    builder.addCase(cakeActions.ordered, (state) =>{
        state.numOfIcecreams--
    })
},
```

Async Thunk

- **Redux toolkit provides createAsyncThunk functions for creation & dispatching of async actions**
- **createAsyncThunk()** will automatically dispatch lifecycle actions based on the returned promise.
- createAsyncThunk generates **pending**, **fulfilled** & **rejected** action types.
- We can listen to these action types in **extraReducers** property & perform necessary state manipulation based on these actions.
- Redux Toolkit uses **redux-thunk** under the hood to achieve this. We explicitly do not require to install redux-thunk package.

userSlice.js

```
const reduxToolkit = require("@reduxjs/toolkit");
const axios = require("axios");

const { createSlice, createAsyncThunk } = reduxToolkit;

const initialState = {
    loading: false,
    users: [],
    error: ""
}

const fetchUsers = createAsyncThunk("user/fetchusers", () => {
    // createAsyncThunk generates "pending", "fulfilled" & "rejected" as action types.

    // Promise can be in "pending" | "fulfilled" | "rejected" state
    return axios.get("https://jsonplaceholder.typicode.com/userss")
        .then((response) => response.data.map(user => user.id));
})
```

```
const userSlice = createSlice({
    name: "user",
    initialState,
    extraReducers: (builder) => {
        builder.addCase(fetchUsers.pending, (state, action) => {
            state.loading = true;
        })
        builder.addCase(fetchUsers.fulfilled, (state, action) => {
            state.loading = false;
            state.users = action.payload;
            state.error = ""
        })
        builder.addCase(fetchUsers.rejected, (state, action) => {
            state.loading = false;
            state.users = [];
            state.error = action.error.message
        })
    }
})

module.exports = userSlice.reducer;
module.exports.fetchUsers = fetchUsers;
```

In **extraReducers** property, we listened to the action types which were generated by fetchUsers async thunk function.

Part-3 - React Redux Package

Setting Up React Project:

In your React Project, Install below dependencies:
npm install axios
npm install @reduxjs/toolkit
npm install react-redux

Create Slice & Store the same way as we made in Part 2: - Redux Toolkit
But in React, we need to use **“import”** instead of **“require”** while importing dependencies.

Also **“exports”** syntax instead of **“module.exports”** while exporting modules.

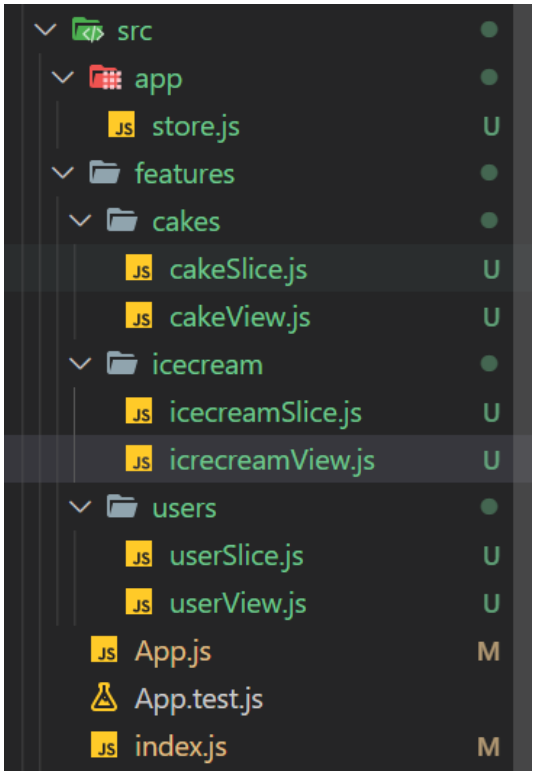
We can then create a view for each slice, where we consume the state and dispatch actions of each slice.

```
1   import React from "react";
2
3   export const CakeView  = () => {
4       return (
5           <div>
6           <h2>Number Of Cakes- </h2>
7           <button>Order Cake</button>
8           <button>Restock Cakes</button>
9           </div>
10      )
11  }
```

App.jsx U

src > JS App.jsx > ...

```jsx
import { useState } from 'react'
import './App.css'
import {CakeView} from './features/cake/CakeView'
import {IcecreamView} from './features/icecream/IcecreamView'
import {UserView} from './features/user/UserView'

function App() {
  const [count, setCount] = useState(0)

  return (
    <div className="App">
      <CakeView />
      <IcecreamView />
      <UserView />
    </div>
  )
}

```

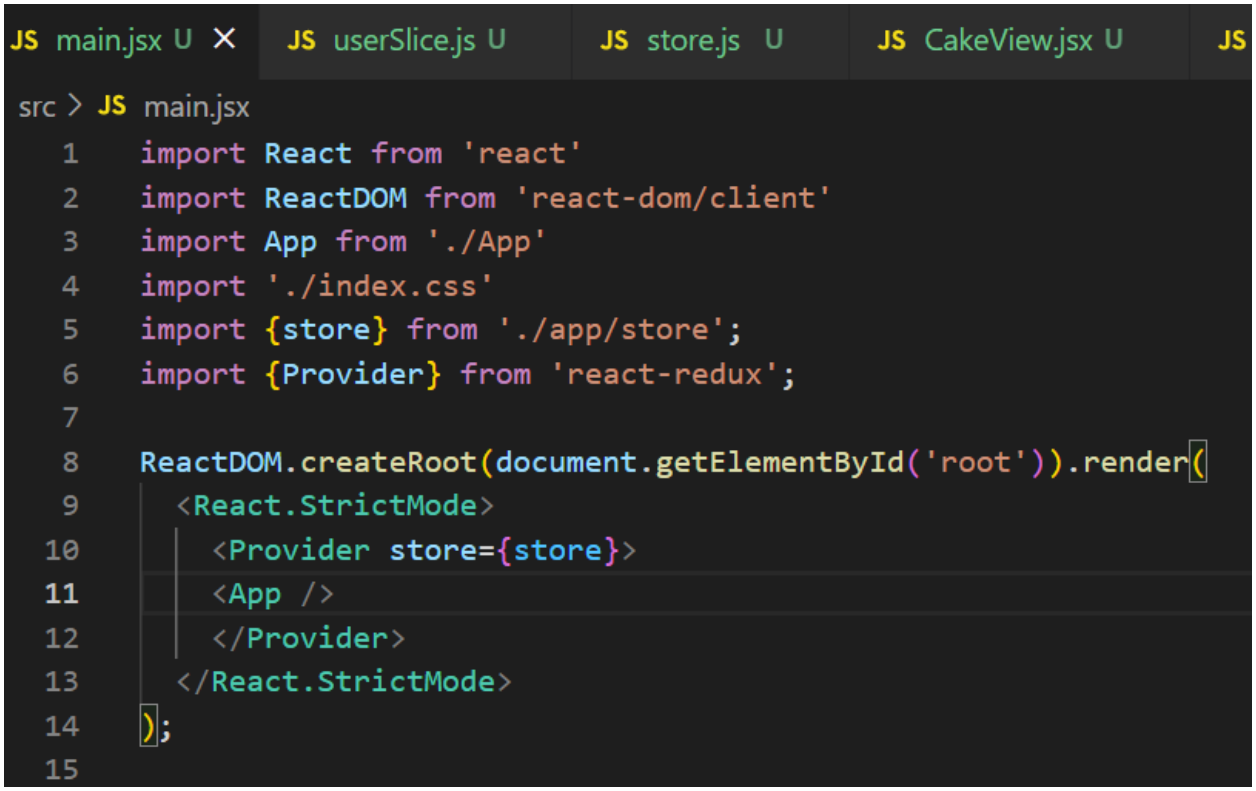


Provider

- After installing the **react-redux** package in our react app. We will use this package to connect redux to our react app.

- For connecting, We will wrap the utmost parent component of react app with <Provider> component provided via “react-redux” package.

- Also we will pass our **store** to this <Provider store = { store }> component **as a prop** so that our complete react app can access the store.

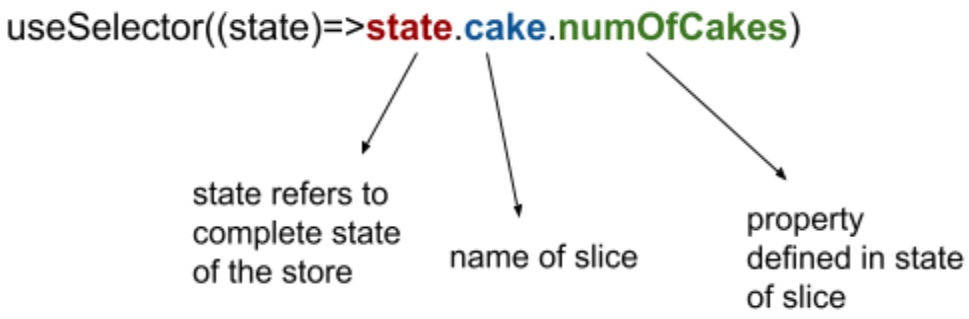


useSelector Hook

It is used to access the state maintained in redux store.

useSelector hook takes function as an input and return of that function is the output of useSelector Hook.

Function which we pass to useSelector hook has **“state”** as an argument and we can return the state as output of useSelector hook.




```jsx
import { useSelector } from "react-redux";

const CakeView = (params) => {

    const numberOfCakes = useSelector((state)=>state.cake.numOfCakes)

    return <>
        <h1>CakeView View</h1>
        <h3>Number of Cakes = {numberOfCakes}</h3>
        <button>Order</button>
        <button>Restock</button>
    </>
}

export default CakeView;
```

useDispatch Hook

This hook is used for dispatching an action.

We can import actions from a slice in the file where we want to dispatch the actions.

```
import { useSelector, useDispatch } from "react-redux";
import { ordered, restocked } from "./cakeSlice";

const CakeView = (params) => {
    const dispatch = useDispatch()
    const numberOfCakes = useSelector((state) => state.cake.numOfCakes)

    return <>
        <h1>CakeView View</h1>
        <h3>Number of Cakes = {numberOfCakes}</h3>
        <button onClick={() => dispatch(ordered(1))}>Order</button>
        <button onClick={() => dispatch(restocked(1))}>Restock</button>
    </>
}

export default CakeView;
```

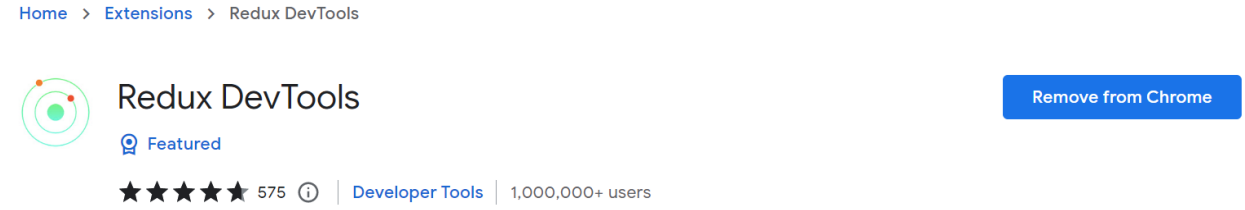
Note:

- It's not mandatory to manage all the states in the Redux store only.
- We can create state (useState) in local components as well, based on our use-case and can pass necessary data to store.

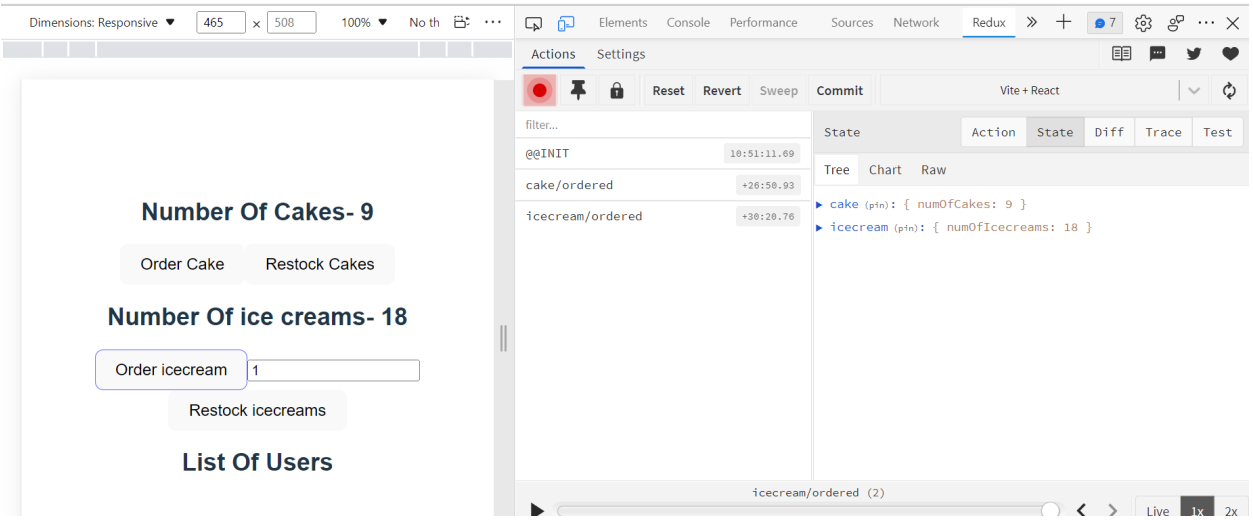
Redux DevTools

We can use Redux DevTools for debugging and analyzing the state transitions without installing any dependencies or npm packages in our project.

We can install it from chrome.

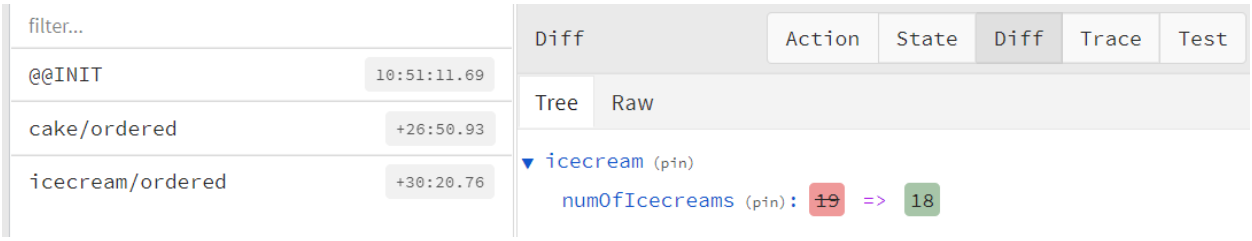


Open Console => Go to **Redux Tab**

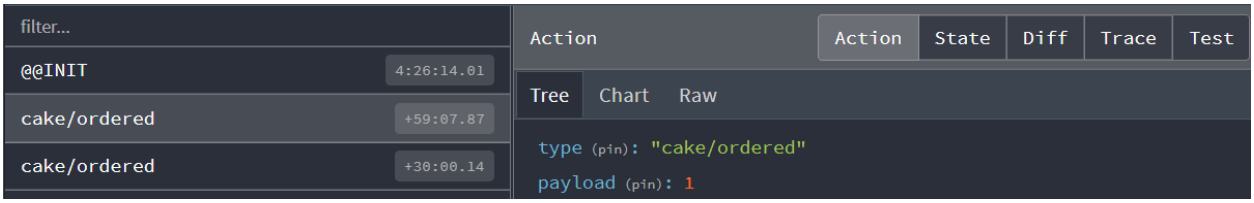


On the left hand side, we can see the actions dispatched.
On right hand, We can view **State** and difference in state with respect to current operation.

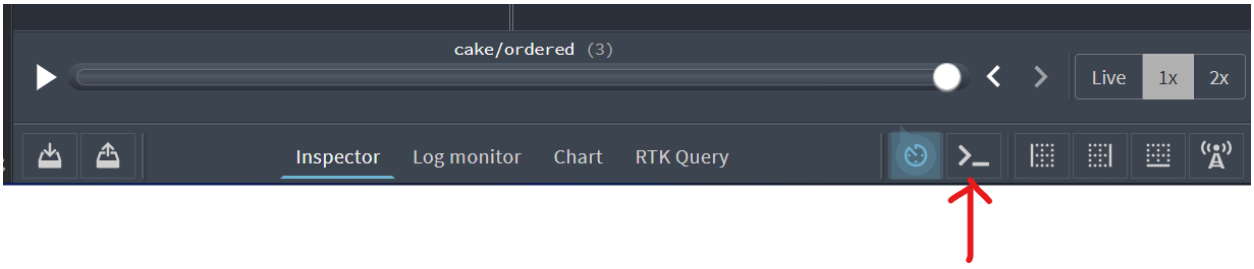
On Right Side, In **Diff** tab, we can see the state transition



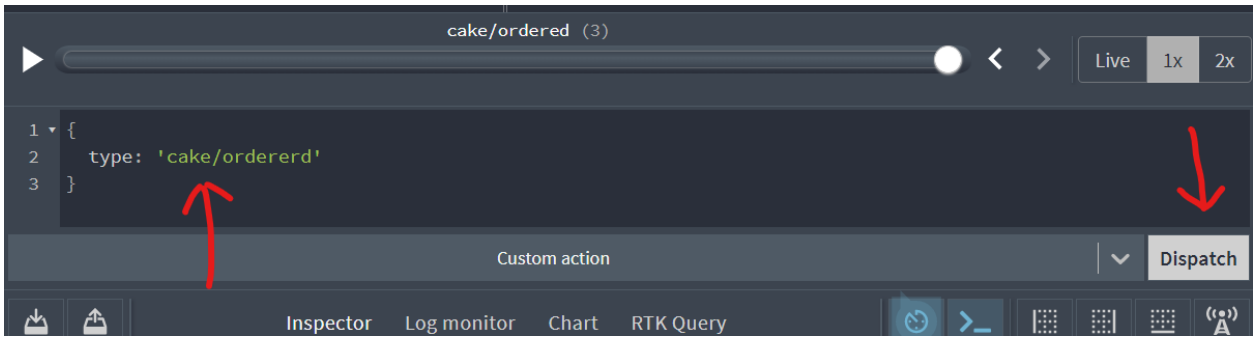
On Right Side, In **Action** Tab, We can check the dispatched action payload



Action Dispatcher Button



We have a **Dispatcher** option available to dispatch the actions through the tool without using React App.

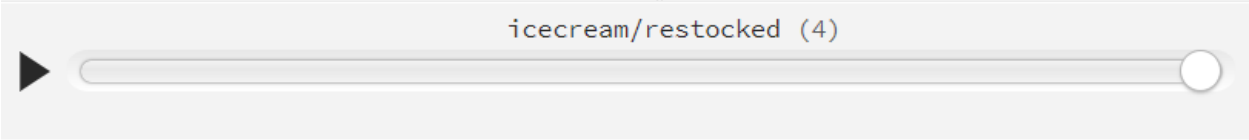


We can manually type the “action” object & click on the “Dispatch” button to manipulate the state and changes are reflected in the UI.

This is especially helpful when the element is hidden in the UI and you have to perform quite a few clicks to perform the right actions.

Ex. Here in the above picture, We have written a cake/ordered in the type and num of cakes got reduced by 1 after clicking on the dispatch option.

Slider



This **slider** cycles through all the actions dispatched so far. The UI also updates at the same time. You can pause at any point of time to verify the state of what it is supposed to be. It is called **time travel debugging**.

Fetching Data via API

Slice Code

```
import { createSlice, createAsyncThunk } from "@reduxjs/toolkit";
import axios from "axios";


const initialState = {
    loading: false,
    users: [],
    error: ""
}

const fetchUsers = createAsyncThunk("user/fetchusers", () => {
    // createAsyncThunk generates "pending", "fulfilled" & "rejected" as action types.

    // Promise can be in "pending" | "fulfilled" | "rejected" state
    return axios.get("https://jsonplaceholder.typicode.com/users")
        .then((response) => response.data.map(user => user.id));
})
```

```
const userSlice = createSlice({
    name: "user",
    initialState,
    extraReducers: {
        [fetchUsers.pending]: (state, action) => {
            state.loading = true;
        },
        [fetchUsers.fulfilled]: (state, action) => {
            state.loading = false;
            state.users = action.payload;
            state.error = ""
        },
        [fetchUsers.rejected]: (state, action) => {
            state.loading = false;
            state.users = [];
            state.error = action.error.message
        }
    }
})

export default userSlice.reducer;
export { fetchUsers }
```

View Code

```jsx
import { useSelector, useDispatch } from "react-redux";
import { fetchUsers } from "./userSlice";

const UserView = (params) => {
    const { loading, users, error } = useSelector((state) => state.user)
    const dispatch = useDispatch();

    return <>
        <h1>Users List</h1>
        <button onClick={() => dispatch(fetchUsers())}>Fetch Users</button>

        {loading && <div> Loading...</div>}
        {error && <div> {error}</div>}
        {users.length ? users.map(id => <div>{id}</div>) : ""}
    </>
}

export default UserView;
```