**React Redux**

1. **Introduction**
   1. **Introduction**

Redux is an open-source JavaScript library for managing the state of an application. It is commonly used in web applications built with frameworks such as React, Angular, or Vue.js.

Redux provides a predictable state container for managing an application's state in a single place, which makes it easier to reason about and debug. It follows a unidirectional data flow pattern, which means that the state of an application can only be modified by dispatching actions to a central store, which contains the application's state. The state is then updated by a pure function called a reducer, which takes the previous state and the dispatched action as input and returns a new state.

Redux also provides a middleware system, which allows developers to customize the behavior of dispatching actions and handling asynchronous operations, such as network requests. It can also be used with tools such as Redux DevTools to debug and analyze the state of an application.

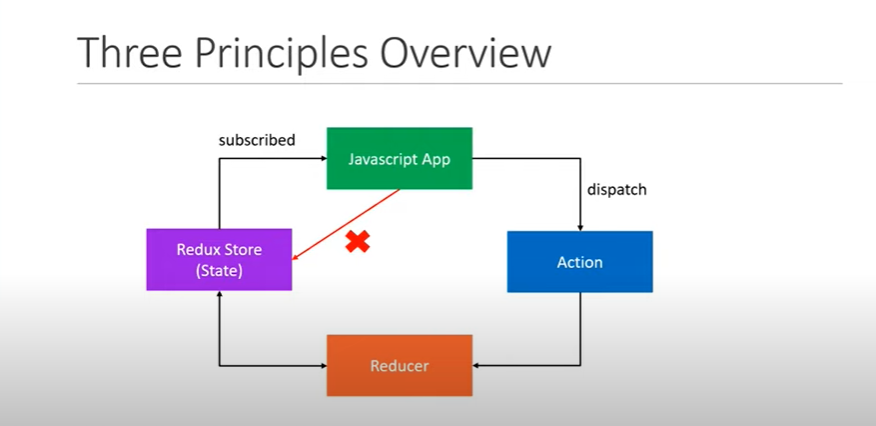
Overall, Redux provides a powerful and flexible way to manage the state of an application, and it is widely used in the JavaScript community for building scalable and maintainable web applications.

* 1. **Getting Started**
* **We create a folder (REDUX-DEMO)**
* **We run the cmd npm init –yes for create our packaje.json file**
* **We insall Redux npm I redux**
* **We add our index.js**
  1. **Three Core Concepts**

Une image contenant table

Description générée automatiquement

* 1. **Three Principles**

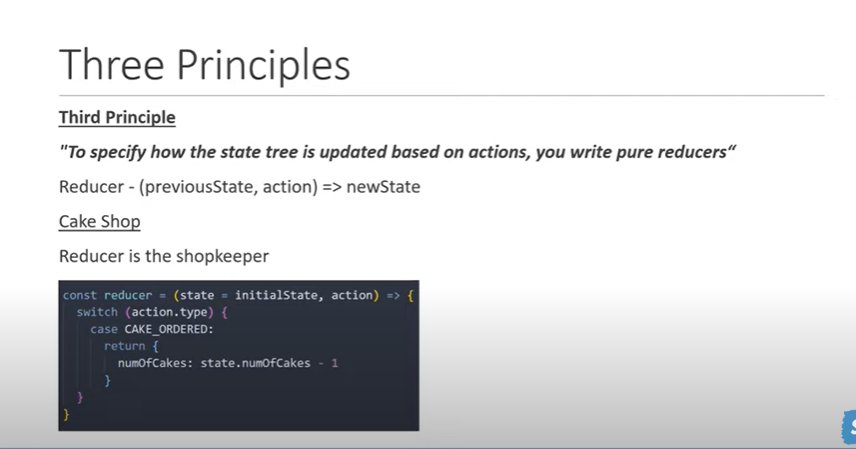


Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement



1. **Lm39oule**
   1. **Action**

In Redux, an action is a plain JavaScript object that represents an intention to change the state of an application. Actions are dispatched to a central store and are the only way to update the state in Redux.

An action object typically has a **type** property that describes the type of action being performed. The **type** property is usually a string constant that is defined in a separate module or file, to ensure consistency across the application. Other properties of an action object can contain additional data that is required to update the state. For example, an action that adds a new item to a todo list might have a **payload** property that contains the text of the new item.

Here is an example of an action object in Redux:

{

    type: 'ADD\_TODO',

    payload: {

      text: 'Buy milk'

    }

  }

To dispatch an action in Redux, you call the **dispatch** function with an action object as an argument, like this:

dispatch({

    type: 'ADD\_TODO',

    payload: {

      text: 'Buy milk'

    }

  });

When an action is dispatched, it is passed to the store's reducer function, which calculates the new state based on the current state and the action. The updated state is then returned from the reducer function and becomes the new state of the application.

* 1. **Reducers**

In Redux, a reducer is a pure function that takes the current state and an action as input, and returns a new state based on the action. Reducers are responsible for calculating the new state of the application in response to dispatched actions.

A reducer function typically has a switch statement that checks the **type** property of the action and performs a specific operation based on the action type. Each case in the switch statement returns a new state that reflects the change caused by the action. The default case should return the current state if the action type is not recognized.

Here is an example of a reducer function in Redux:

function todoReducer(state = initialState, action) {

    switch (action.type) {

      case 'ADD\_TODO':

        return {

          ...state,

          todos: [

            ...state.todos,

            {

              id: Date.now(),

              text: action.payload.text,

              completed: false

            }

          ]

        };

      case 'TOGGLE\_TODO':

        return {

          ...state,

          todos: state.todos.map(todo => {

            if (todo.id === action.payload.id) {

              return {

                ...todo,

                completed: !todo.completed

              };

            }

            return todo;

          })

        };

      default:

        return state;

    }

  }

In this example, the **todoReducer** function takes the current state (which has a **todos** property that is an array of todo items) and an action as input. If the action type is **ADD\_TODO**, the reducer returns a new state that adds a new todo item to the **todos** array. If the action type is **TOGGLE\_TODO**, the reducer returns a new state that toggles the **completed** property of a specific todo item. The default case returns the current state if the action type is not recognized.

Reducers should be pure functions, which means they should not modify the original state or the action object, and they should not have any side effects. Instead, reducers should always return a new state object that represents the updated state of the application.

* 1. **Store**

In Redux, a store is an object that holds the current state of the application, and provides methods to update the state and subscribe to changes. The store is the single source of truth for the state of the application, and is responsible for dispatching actions to reducers and updating the state based on the reducers' output.

The store is created by calling the **createStore** function from the Redux library, and passing in a reducer function as an argument. The **createStore** function returns a store object that has three methods:

1. **getState()** - Returns the current state of the store.
2. **dispatch(action)** - Dispatches an action to the store, which triggers the reducer function to calculate the new state.
3. **subscribe(listener)** - Registers a listener function that is called whenever the state of the store changes.

Here is an example of creating a store in Redux:

import { createStore } from 'redux';

import todoReducer from './reducers/todoReducer';

const store = createStore(todoReducer);

In this example, the **createStore** function is used to create a store object that uses the **todoReducer** function as its reducer. The **store** object is then available for use in the application, and can be accessed using the **getState**, **dispatch**, and **subscribe** methods.

The store in Redux is typically a top-level object in the application, and is often passed down to child components through the use of React's **Context API** or other similar techniques. This allows components to access and update the state of the application using the methods provided by the store object.

Une image contenant texte

Description générée automatiquement

**Example:**

const redux = require('redux')

const CAKE\_ORDERED = "CAKE\_ORDERED";

function orderCake() {

  return {

    type: CAKE\_ORDERED,

    quantity: 1,

  };

}

const initialState = {

  numOfCakes: 10,

};

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

  switch (action.type) {

    case CAKE\_ORDERED:

      return { ...state, numOfCakes: state.numOfCakes - 1 }

      default:return state

  }

};

const store=redux.createStore(reducer)

console.log("Initial state", store.getState())

const unsubscribe= store.subscribe(()=>{

    console.log('Update state',store.getState())

})

store.dispatch(orderCake())

store.dispatch(orderCake())

store.dispatch(orderCake())

unsubscribe()

Une image contenant texte

Description générée automatiquement

* 1. **Just an Example for using payload.**

const redux = require("redux");

const CAKE\_ORDERED = "CAKE\_ORDERED";

const CAKE\_RESTOCKED = "CAKE\_RESTOCKED";

function orderCake() {

  return {

    type: CAKE\_ORDERED,

    payload: 1,

  };

}

function restockCake(qty = 1) {

  return {

    type: CAKE\_RESTOCKED,

    payload: qty,

  };

}

const initialState = {

  numOfCakes: 10,

};

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

  switch (action.type) {

    case CAKE\_ORDERED:

      return { ...state, numOfCakes: state.numOfCakes - 1 };

    case CAKE\_RESTOCKED:

      return {

        ...state,

        numOfCakes: state.numOfCakes + action.payload,

      };

    default:

      return state;

  }

};

const store = redux.createStore(reducer);

console.log("Initial state", store.getState());

const unsubscribe = store.subscribe(() => {

  console.log("Update state", store.getState());

});

store.dispatch(orderCake());

store.dispatch(orderCake());

store.dispatch(orderCake());

store.dispatch(restockCake(4))

unsubscribe();

Une image contenant texte

Description générée automatiquement

1. **Lm39oule2**
   1. **Multiple Reducers**

Une image contenant texte

Description générée automatiquement

In Redux, a reducer is a pure function that takes the current state and an action, and returns the new state based on that action. Redux applications typically have a single reducer function that manages the entire state of the application. However, as the application grows, it may become more complex, and managing all of the state in a single reducer can become cumbersome.

Using multiple reducers in Redux can help to organize the state of the application, making it easier to manage and understand. This can also make it easier to work in a team, as different members of the team can work on different parts of the application's state.

Redux provides a built-in function called **combineReducers** that allows you to combine multiple reducers into a single reducer function. This function takes an object whose values are individual reducer functions, and returns a new reducer function that you can use to manage the state of the entire application.

Using multiple reducers can also help with performance, as it allows you to update only the part of the state that is affected by a given action, rather than having to update the entire state every time an action is dispatched.

In summary, using multiple reducers in Redux can help with organization, maintainability, teamwork, and performance, especially as the application becomes more complex.

**CombineReducers**

const rootReducer = redux.combineReducers({

    cake:Cakereducer,

    iceCream:icacreamReducer

})

const store=redux.createStore(rootReducer)

**Example:**

const redux = require("redux");

const CAKE\_ORDERED = "CAKE\_ORDERED";

const CAKE\_RESTOCKED = "CAKE\_RESTOCKED";

const ICECREAM\_ORDERED = 'ICECREAM\_ORDERED'

const ICECREAM\_RESTOCKED = 'ICECREAM\_RESTOCKED'

function orderCake() {

  return {

    type: CAKE\_ORDERED,

    payload: 1,

  };

}

function restockCake(qty = 1) {

  return {

    type: CAKE\_RESTOCKED,

    payload: qty,

  };

}

function orderIceCream(qty = 1) {

    return {

      type: ICECREAM\_ORDERED,

      payload: qty

    }

  }

  function restockIceCream(qty = 1) {

    return {

      type: ICECREAM\_RESTOCKED,

      payload: qty

    }

  }

// const initialState = {

//   numOfCakes: 10,

//   numOfIceCreams: 20

// };

const initialCakeState = {

    numOfCakes: 10

  }

  const initialIceCreamState = {

    numOfIceCreams: 20

  }

const Cakereducer = (state = initialCakeState, action) => {

  switch (action.type) {

    case ICECREAM\_ORDERED:

      return { ...state, numOfCakes: state.numOfCakes - 1 };

    case ICECREAM\_RESTOCKED:

      return {

        ...state,

        numOfCakes: state.numOfCakes + action.payload,

      };

    default:

      return state;

  }

};

const icacreamReducer = (state = initialCakeState, action) => {

    switch (action.type) {

      case CAKE\_ORDERED:

        return { ...state, numOfCakes: state.numOfCakes - 1 };

      case CAKE\_RESTOCKED:

        return {

          ...state,

          numOfCakes: state.numOfCakes + action.payload,

        };

      default:

        return state;

    }

  };

// const store = redux.createStore(icacreamReducer);

const rootReducer = redux.combineReducers({

    cake:Cakereducer,

    iceCream:icacreamReducer

})

const store=redux.createStore(rootReducer)

console.log("Initial state", store.getState());

const unsubscribe = store.subscribe(() => {

  console.log("Update state", store.getState());

});

store.dispatch(orderCake());

store.dispatch(orderCake());

store.dispatch(orderCake());

store.dispatch(restockCake(4))

store.dispatch(orderIceCream());

store.dispatch(orderIceCream());

store.dispatch(orderIceCream());

store.dispatch(restockIceCream(4))

unsubscribe();

Une image contenant texte

Description générée automatiquement

* 1. **Immer**

Immer is a JavaScript library that simplifies the process of working with immutable data structures. It allows you to write code that looks like you are directly manipulating mutable data, but under the hood, it creates new immutable data structures. Immer is often used with state management libraries such as Redux or MobX, but it can be used in any JavaScript project that needs to work with immutable data.

With Immer, you can create a "draft" of an object, which is a mutable copy of the original object. You can then modify the draft using familiar mutation syntax, such as assignment or property deletion. Immer applies those modifications to a new immutable copy of the original object, so the original object remains unchanged.

const redux = require('redux')

const produce = require('immer').produce

const initialState = {

  name: 'Vishwas',

  address: {

    street: '123 Main St',

    city: 'Boston',

    state: 'MA'

  }

}

const STREET\_UPDATED = 'STREET\_UPDATED'

const updateStreet = street => {

  return {

    type: STREET\_UPDATED,

    payload: street

  }

}

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

  switch (action.type) {

    case STREET\_UPDATED:

      //   return {

      //     ...state,

      //     address: {

      //       ...state.address,

      //       street: action.payload

      //     }

      //   }

      return produce(state, draft => {

        draft.address.street = action.payload

      })

    default: {

      return state

    }

  }

}

const store = redux.createStore(reducer)

console.log('Initial State ', store.getState())

const unsubscribe = store.subscribe(() => {

  console.log('Updated State ', store.getState())

})

store.dispatch(updateStreet('456 Main St'))

unsubscribe()

* 1. **Middleware**

In the context of software, middleware refers to software that acts as a bridge between different components or applications. In web development, middleware is commonly used to process HTTP requests and responses, and it sits between the client and the server.

In the context of the Redux state management library, middleware is a way to intercept and modify the behavior of the Redux dispatch function, which is responsible for dispatching actions to the reducers. Middleware can be used to perform tasks such as logging, error handling, and asynchronous data fetching.

In a Redux application, middleware functions are functions that take the dispatch function as an argument, and return another function that takes the next middleware function or the dispatch function itself as an argument. The returned function can then intercept and modify the action and/or the state before passing it on to the next middleware function or the dispatch function.

* + 1. **Middleware and logging**
* For te logger we need to install 🡺 npm i redux-logger

const redux = require("redux");

const reduxLogger = require('redux-logger')

const CAKE\_ORDERED = "CAKE\_ORDERED";

const CAKE\_RESTOCKED = "CAKE\_RESTOCKED";

const ICECREAM\_ORDERED = 'ICECREAM\_ORDERED'

const ICECREAM\_RESTOCKED = 'ICECREAM\_RESTOCKED'

function orderCake() {

  return {

    type: CAKE\_ORDERED,

    payload: 1,

  };

}

function restockCake(qty = 1) {

  return {

    type: CAKE\_RESTOCKED,

    payload: qty,

  };

}

function orderIceCream(qty = 1) {

    return {

      type: ICECREAM\_ORDERED,

      payload: qty

    }

  }

  function restockIceCream(qty = 1) {

    return {

      type: ICECREAM\_RESTOCKED,

      payload: qty

    }

  }

// const initialState = {

//   numOfCakes: 10,

//   numOfIceCreams: 20

// };

const initialCakeState = {

    numOfCakes: 10

  }

  const initialIceCreamState = {

    numOfIceCreams: 20

  }

const Cakereducer = (state = initialCakeState, action) => {

  switch (action.type) {

    case ICECREAM\_ORDERED:

      return { ...state, numOfCakes: state.numOfCakes - 1 };

    case ICECREAM\_RESTOCKED:

      return {

        ...state,

        numOfCakes: state.numOfCakes + action.payload,

      };

    default:

      return state;

  }

};

const icacreamReducer = (state = initialCakeState, action) => {

    switch (action.type) {

      case CAKE\_ORDERED:

        return { ...state, numOfCakes: state.numOfCakes - 1 };

      case CAKE\_RESTOCKED:

        return {

          ...state,

          numOfCakes: state.numOfCakes + action.payload,

        };

      default:

        return state;

    }

  };

// const store = redux.createStore(icacreamReducer);

const rootReducer = redux.combineReducers({

    cake:Cakereducer,

    iceCream:icacreamReducer

})

const store=redux.createStore(rootReducer,redux.applyMiddleware(reduxLogger.createLogger()))

console.log("Initial state", store.getState());

const unsubscribe = store.subscribe(() => {

//   console.log("Update state", store.getState());

});

store.dispatch(orderCake());

store.dispatch(orderCake());

store.dispatch(orderCake());

store.dispatch(restockCake(4))

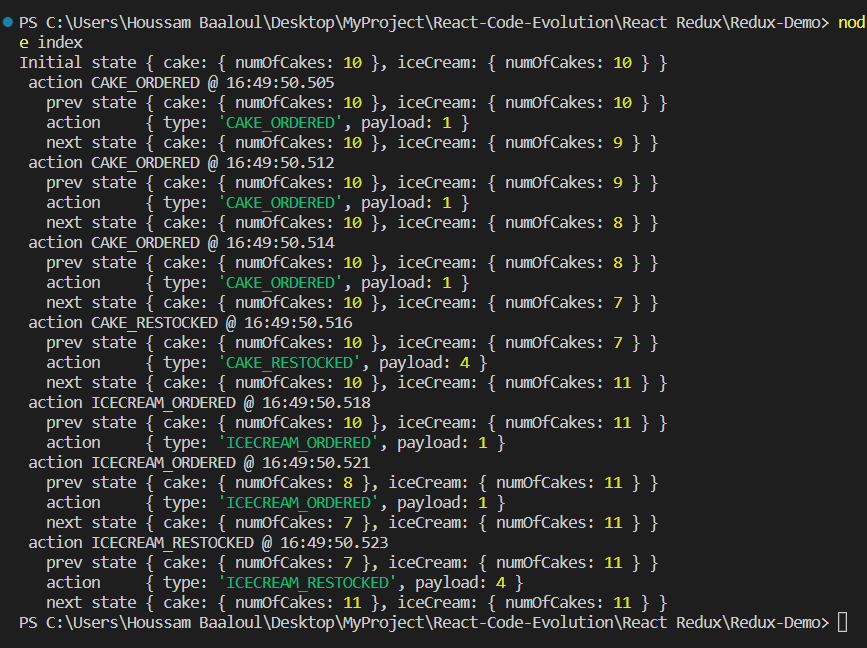
store.dispatch(orderIceCream());

store.dispatch(orderIceCream());

store.dispatch(orderIceCream());

store.dispatch(restockIceCream(4))

unsubscribe();



* + 1. Async Actions

In Redux, an asynchronous action is an action that does not immediately return a value, but instead dispatches another action after some asynchronous operation has completed.

For example, an asynchronous action might be used to fetch data from an API endpoint, wait for the response to be received, and then dispatch another action with the data. In this case, the first action would initiate the API call and the second action would update the application state with the data received from the API.

There are several ways to implement asynchronous actions in Redux. One common approach is to use middleware, such as Redux Thunk or Redux Saga, to handle asynchronous operations. Middleware provides a way to intercept actions before they are passed on to the reducers, and can be used to perform asynchronous operations before dispatching additional actions.

In summary, asynchronous actions in Redux are actions that dispatch other actions after some asynchronous operation has completed, and are commonly implemented using middleware.

* + - 1. Async Action and fetching data.
* We need to install redux-thunk and axios 🡺 **npm i axios redux-thunk**

const redux = require('redux')

const thunkMiddleware = require('redux-thunk').default

const axios = require('axios')

const initialState = {

    loading: false,

    users: [],

    error: ''

  }

  const FETCH\_USERS\_REQUESTED = 'FETCH\_USERS\_REQUESTED'

  const FETCH\_USERS\_SUCCEEDED = 'FETCH\_USERS\_SUCCEEDED'

  const FETCH\_USERS\_FAILED = 'FETCH\_USERS\_FAILED'

  const fetchUsersRequest = () => {

    return {

      type: FETCH\_USERS\_REQUESTED

    }

  }

  const fetchUsersSuccess = users => {

    return {

      type: FETCH\_USERS\_SUCCEEDED,

      payload: users

    }

  }

  const fetchUsersFailure = error => {

    return {

      type: FETCH\_USERS\_FAILED,

      payload: error

    }

  }

  const fetchUsers = () => {

    return function (dispatch) {

      dispatch(fetchUsersRequest())

      axios

        .get('https://jsonplaceholder.typicode.com/users')

        .then(response => {

          // response.data is the users

          const users = response.data.map(user => user.id)

          dispatch(fetchUsersSuccess(users))

        })

        .catch(error => {

          // error.message is the error message

          dispatch(fetchUsersFailure(error.message))

        })

    }

  }

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

    console.log(action.type)

    switch (action.type) {

      case FETCH\_USERS\_REQUESTED:

        return {

          ...state,

          loading: true

        }

      case FETCH\_USERS\_SUCCEEDED:

        return {

          loading: false,

          users: action.payload,

          error: ''

        }

      case FETCH\_USERS\_FAILED:

        return {

          loading: false,

          users: [],

          error: action.payload

        }

    }

  }

  const store=redux.createStore(reducer,redux.applyMiddleware(thunkMiddleware))

  store.subscribe(() => {

    console.log(store.getState())

  })

  store.dispatch(fetchUsers())

Une image contenant texte

Description générée automatiquement

1. **Redux Toolkit**
   1. **Definition and setup**

Redux Toolkit is a set of utilities and abstractions that makes it easier to work with Redux, a popular state management library for JavaScript applications. Redux Toolkit is designed to simplify many common use cases for Redux, including writing reducers, creating actions, and configuring the store.

Here are some of the key features of Redux Toolkit:

1. Simplified Reducer Syntax: Redux Toolkit includes a **createSlice** function that reduces the amount of boilerplate code required to create a reducer. It automatically generates action creators and uses an Immer library to allow you to write simpler, immutable code that updates your state.
2. Default Configuration: Redux Toolkit includes a default configuration that sets up your store with commonly used middleware like **redux-thunk** and **redux-logger**.
3. Immutable State Management: Redux Toolkit uses the Immer library to allow you to write simpler, mutable code that updates your state. This means that you can write code that modifies state directly, without worrying about immutability.
4. Thunk Simplification: Redux Toolkit provides a simplified syntax for creating thunk functions that make it easier to handle asynchronous actions.
5. Additional Utilities: Redux Toolkit includes a number of additional utilities, such as **createAsyncThunk** for handling asynchronous actions, and **createEntityAdapter** for managing normalized data.

Overall, Redux Toolkit can help you write cleaner, more concise, and more maintainable Redux code. It's a great choice for developers who want to use Redux, but don't want to spend a lot of time writing boilerplate code.

Une image contenant texte, oiseau aquatique, capture d’écran

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Setup:🡺 **npm i @reduxjs/toolkit**

* 1. **createSlice**

**createSlice** is a function provided by Redux Toolkit, a utility library for Redux, that helps you create a Redux slice. A slice is a portion of your Redux store that contains a single reducer function and a set of actions that operate on a particular subset of your application state.

**createSlice** reduces the amount of boilerplate code required to create a reducer by generating the action creators and action types automatically. Here's how it works:

cakeSlice.js

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

const initialState = {

  numOfCakes: 10,

};

const cakeSlice = reduxToolkit.createSlice({

  name: "cake",

  initialState,

  reducers: {

    ordered: (state) => {

      state.numOfCakes--;

    },

    restocked: (state, action) => {

      state.numOfCakes += action.payload;

    },

  },

});

module.exports.cakeReducer = cakeSlice.reducer;

module.exports.cakeActions = cakeSlice.actions;

* + 1. **Configure store.**

**configureStore** is a function provided by Redux Toolkit, a utility library for Redux, that helps you configure a Redux store with sensible defaults and middleware. It's a wrapper around the Redux **createStore** function, but with additional features and optimizations.

**Store.js**

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

const { cakeReducer } = require("../features/cake/cakeSlice");

const store=reduxToolkit.configureStore({

    reducer:{

        cake:cakeReducer

    }

})

module.exports.store=store

**index.js**

const {store} = require('./app/store')

const { cakeActions } = require('./features/cake/cakeSlice')

console.log('initial state',store.getState())

const unsubscribe=store.subscribe(()=>{

    console.log("Update state",store.getState())

})

store.dispatch(cakeActions.ordered())

store.dispatch(cakeActions.ordered())

store.dispatch(cakeActions.ordered())

store.dispatch(cakeActions.restocked(3))

unsubscribe()

Une image contenant texte, intérieur, capture d’écran

Description générée automatiquement

* + 1. **Multiple reducer (Tres simple modification en store.js)**

**icecreamSlice.js**

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

const initialState = {

  numOfIcecreams: 10,

};

const icecreamSlice = reduxToolkit.createSlice({

  name: "icecream",

  initialState,

  reducers:{

    ordered:(state)=>{

        state.numOfIcecreams--

    },

    restocked:(state,action)=>{

        state.numOfIcecreams+=action.payload

    }

  }

});

module.exports.icecreamReducer=icecreamSlice.reducer

module.exports.icecreamActions=icecreamSlice.actions

**store.js**

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

const { cakeReducer } = require("../features/cake/cakeSlice");

const { icecreamReducer } = require("../features/icecream/icecreamSlice");

const store=reduxToolkit.configureStore({

    reducer:{

        cake:cakeReducer,

        icecream:icecreamReducer

    }

})

module.exports.store=store

**index.js**

const {store} = require('./app/store')

const { cakeActions } = require('./features/cake/cakeSlice')

const { icecreamActions } = require('./features/icecream/icecreamSlice')

console.log('initial state',store.getState())

const unsubscribe=store.subscribe(()=>{

    console.log("Update state",store.getState())

})

store.dispatch(cakeActions.ordered())

store.dispatch(cakeActions.ordered())

store.dispatch(cakeActions.ordered())

store.dispatch(cakeActions.restocked(3))

store.dispatch(icecreamActions.ordered())

store.dispatch(icecreamActions.ordered())

store.dispatch(icecreamActions.restocked(6))

unsubscribe()

Une image contenant texte

Description générée automatiquement

* 1. **Logger Middleware**
* **npm I redux-logger**

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

const { cakeReducer } = require("../features/cake/cakeSlice");

const { icecreamReducer } = require("../features/icecream/icecreamSlice");

const reduxLogger=require('redux-logger')

const logger=reduxLogger.createLogger()

const store=reduxToolkit.configureStore({

    reducer:{

        cake:cakeReducer,

        icecream:icecreamReducer

    },

    middleware:(getDefaultMiddleware)=>getDefaultMiddleware().concat(logger)

})

module.exports.store=store

Une image contenant texte

Description générée automatiquement

* 1. **ExtraReducers**

"Extra reducers" is not a term commonly used in the context of Redux or React. However, based on the term "reducers," I assume you are asking about Redux reducers.

In Redux, a reducer is a pure function that takes the current state and an action as input and returns a new state. Reducers are used to update the application state in response to actions dispatched by the application.

An extra reducer can refer to a reducer function that is not part of the main reducer function exported by a Redux module. In other words, it is a separate reducer function that can be imported and combined with other reducers in the Redux store configuration.

**icecreamSlice.js**

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

const initialState = {

  numOfIcecreams: 10,

};

const icecreamSlice = reduxToolkit.createSlice({

  name: "icecream",

  initialState,

  reducers:{

    ordered:(state)=>{

        state.numOfIcecreams--

    },

    restocked:(state,action)=>{

        state.numOfIcecreams+=action.payload

    }

  },

  extraReducers:{

    ['cake/ordered']:(state)=>{

        state.numOfIcecreams-=2

    }

  }

});

module.exports.icecreamReducer=icecreamSlice.reducer

module.exports.icecreamActions=icecreamSlice.actions

* 1. **Async Thunks**

**createAsyncThunk** is a function provided by Redux Toolkit, a library that simplifies the management of state in Redux applications. **createAsyncThunk** is a utility function that creates an async action creator that dispatches three different actions based on the status of the asynchronous operation: **pending**, **fulfilled**, and **rejected**.

Here's an example of how to use **createAsyncThunk**:

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

import { fetchUsers } from './api';

export const fetchUsersAsync = createAsyncThunk(

  'users/fetchUsers',

  async () => {

    const response = await fetchUsers();

    return response.data;

  }

);

In this example, we're creating an async action creator called **fetchUsersAsync** using **createAsyncThunk**. The first argument to **createAsyncThunk** is a string that defines the name of the action. The second argument is an async function that performs the asynchronous operation. In this case, we're using an async function to fetch data from an API using **fetchUsers** and returning the data.

**createAsyncThunk** returns a thunk function that can be dispatched like any other action creator. When the async function completes, **createAsyncThunk** will dispatch the appropriate action based on the result:

* **pending**: Dispatched when the async operation starts.
* **fulfilled**: Dispatched when the async operation completes successfully, with the result of the operation as the action payload.
* **rejected**: Dispatched when the async operation fails, with the error message as the action payload.

You can listen for these actions in your reducers and update the state of your application accordingly.

Using **createAsyncThunk** can simplify your code and reduce the boilerplate required to handle asynchronous actions in your Redux application.

**userSlice.js**

const axios = require('axios')

const createSlice = require('@reduxjs/toolkit').createSlice

const createAsyncThunk = require('@reduxjs/toolkit').createAsyncThunk

const initialState = {

  loading: false,

  users: [],

  error: ''

}

// Generates pending, fulfilled and rejected action types

const fetchUsers = createAsyncThunk('user/fetchUsers', () => {

  return axios

    .get('https://jsonplaceholder.typicode.com/users')

    .then(response => response.data.map(user => user.id))

})

const userSlice = createSlice({

  name: 'user',

  initialState,

  extraReducers: builder => {

    builder.addCase(fetchUsers.pending, state => {

      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.userRecer = userSlice.reducer

module.exports.fetchUsers = fetchUsers

**store.js**

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

const { cakeReducer } = require("../features/cake/cakeSlice");

const { icecreamReducer } = require("../features/icecream/icecreamSlice");

const reduxLogger=require('redux-logger');

const { userRecer } = require("../features/user/userSlice");

const logger=reduxLogger.createLogger()

const store=reduxToolkit.configureStore({

    reducer:{

        cake:cakeReducer,

        icecream:icecreamReducer,

        user:userRecer

    },

    // middleware:(getDefaultMiddleware)=>getDefaultMiddleware().concat(logger)

})

module.exports.store=store

**index.js**

const {store} = require('./app/store')

const { cakeActions } = require('./features/cake/cakeSlice')

const { icecreamActions } = require('./features/icecream/icecreamSlice')

const { fetchUsers } = require('./features/user/userSlice')

console.log('initial state',store.getState())

const unsubscribe=store.subscribe(()=>{

    console.log("Update state",store.getState())

})

// store.dispatch(cakeActions.ordered())

// store.dispatch(cakeActions.ordered())

// store.dispatch(cakeActions.ordered())

// store.dispatch(cakeActions.restocked(3))

// store.dispatch(icecreamActions.ordered())

// store.dispatch(icecreamActions.ordered())

// store.dispatch(icecreamActions.restocked(6))

// unsubscribe()

store.dispatch(fetchUsers())

Une image contenant texte

Description générée automatiquement

1. **React-redux**

**In this section we will use the same folders( feature , app) and we past in src folder**

* 1. **Provider**

In the context of React Redux, a provider is a component that provides the Redux store to the rest of the application.

The Provider component is imported from the react-redux library, and is typically placed at the top level of the component hierarchy, surrounding the entire app. It accepts a prop called "store", which is the Redux store that is created using the createStore() method.

By wrapping the application with the Provider component and passing in the store as a prop, all child components can access the Redux store using the connect() function from the react-redux library. This allows components to read and update the store's state and dispatch actions to update the state.

**Index.js**

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 { store } from './app/store';

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(

  <React.StrictMode>

    <Provider store={store}>

    <App />

    </Provider>

  </React.StrictMode>

);

// If you want to start measuring performance in your app, pass a function

// to log results (for example: reportWebVitals(console.log))

// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals

reportWebVitals();

* 1. **useSelector**

**useSelector** is a hook provided by the React Redux library that allows you to extract data from the Redux store in a React component.

icecreamView.jsx

import React from "react";

import { useSelector } from "react-redux";

export const IcecreamView = () => {

  const numOfIcecrem = useSelector((state) => state.icecream.numOfIcecreams);

  return (

    <div>

      <div>Number of ice creams:{numOfIcecrem}</div>

      <button>Order ice cream</button>

      <button>Restock ice cream</button>

    </div>

  );

};

* 1. **useDispatch**

**useDispatch** is a hook provided by the React Redux library that allows you to dispatch actions to the Redux store from a React component.

import React from "react";

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

import { ordered, restocked } from "./icecreamSlice";

export const IcecreamView = () => {

  const numOfIcecrem = useSelector((state) => state.icecream.numOfIcecreams);

  const dispatch=useDispatch()

  return (

    <div>

      <div>Number of ice creams:{numOfIcecrem}</div>

      <button onClick={()=>dispatch(ordered())}>Order ice cream</button>

      <button onClick={()=>dispatch(restocked(3))}>Restock ice cream</button>

    </div>

  );

};

* 1. **ReduxDevTools**

Redux DevTools is a browser extension and a development tool for debugging and inspecting Redux applications. It allows you to inspect the state of your Redux store, track the changes to your state over time, and debug your application by replaying actions and stepping through your code.

To use the Redux DevTools, you need to install the browser extension first. Once installed, you can access the DevTools by opening the browser's developer console, and selecting the Redux DevTools tab.

The DevTools extension provides several features to help you debug and inspect your Redux store:

1. Time-travel debugging: Allows you to replay actions and view the state of your application at any point in time. You can step forward and backward through your actions to track the changes to your state.
2. Inspect state: Allows you to view the current state of your Redux store, as well as the previous states of your application.
3. Actions log: Displays a log of all the actions that have been dispatched to your Redux store, along with the time and any relevant data.
4. Dispatch actions: Allows you to dispatch actions to your Redux store directly from the DevTools console, which is useful for testing and debugging your application.

The Redux DevTools can be a powerful tool for debugging and optimizing your Redux application, especially when dealing with complex state management and large data sets.

**All the explanation is in this video:**

[**https://www.youtube.com/watch?v=iECBiELxrNE&list=PLC3y8-rFHvwiaOAuTtVXittwybYIorRB3&index=29&ab\_channel=Codevolution**](https://www.youtube.com/watch?v=iECBiELxrNE&list=PLC3y8-rFHvwiaOAuTtVXittwybYIorRB3&index=29&ab_channel=Codevolution)

* 1. **Fetching data**

import React, { useEffect } from "react";

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

import { fetchUsers } from "./userSlice";

export const UserView = () => {

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

  const dispatch = useDispatch();

  useEffect(() => {

    console.log(user)

    dispatch(fetchUsers());

  }, []);

  return( <div>

    <h2>List of Users</h2>

    {user.loading && <div>Loading...</div>}

    {!user.loading && user.error ?<div>Error:{user.error}</div> : null}

    {!user.loading && user.users.length ?(

<ul>{

  user.users.map(user=>{

  return  <li key={user.id}>{user.name}</li>

  })

  }</ul>

    ):null}

  </div>)

};