**Redux**

Redux is open-source, cross-platform library for managing the application state. Redux solve the problems sharing data between components by keeps a single store (single source of truth) of all our application state and the components can connect to the redux store and get back the data.

Redux follows a unidirectional data flow. Redux has 3 major components “actions”, “reducers” and the “stores”.

Action

Component view

Redux store

Reducer

**Actions**

Actions are simple javascript object. It has a ‘type’ property which specifies the type of the action we are performing and optionally, can also have a “payload” property which is used to send some data to our redux store.

**Reducers**

Reducers are the functions that determine the changes in application state and return the updated state. They take actions as the argument and update the state inside the store.

**Redux middleware**

Redux middleware is a function or a piece of code that sits between action and reducer and can be interact with the dispatched action before reaching the reducer functions.

**Redux thunk**

Redux thunk is a middleware that allows us to call the action creators that return a function(thunk) which takes the store’s dispatch method as the argument and which is afterwards used to dispatch the synchronous action after the api or side effects has been finished.

**Redux saga**

Redux saga is a middleware library that help us with api calls or side effects. Redux saga uses ‘generators’ to write asynchronous code. Generator function is a javascript function that allow us to write asynchronous code in a more synchronous and linear fashion.

import { put, takeEvery } from 'redux-saga/effects';

import { fetchDataSuccess, fetchDataFailure } from './actions';

import { FETCH\_DATA\_REQUEST } from './actionTypes';

function\* fetchDataWorker(action) {

  try {

    const data = yield fetch('/api/data').then(response => response.json());

    yield put(fetchDataSuccess(data));

  } catch (error) {

    yield put(fetchDataFailure(error));

  }

}

function\* watchFetchData() {

  yield takeEvery(FETCH\_DATA\_REQUEST, fetchDataWorker);

}

Generator function

Generator function in javascript is a special type of function that can be paused and resumed. It’s denoted by asterisk ‘\*’ after the function keyword. Generator function are used to create generator objects, which can produce a sequence of value lazily, one at a tim, using the yield keyword. The generator functions execution is paused at each ‘yield’ statement until it’s resumed by calling the next () method on the generator object.

function\* simpleGenerator (){

yield 1;

yield2;

yield3;

}

const generator = simpleGenerator ();

console.log (generator.next ()); // { value: 1, done: false }

console.log (generator.next ()); // { value: 2, done: false }

console.log (generator.next ()); // { value: 3, done: false }

console.log (generator.next()); // { value: undefined, done: true }

**Redux js**

Actions interaction and working

Component, both functional and class-based, can dispatch actions. When a user interacts with ui element, such as clicking a button, a component can trigger an action by calling a function that dispatches the action to the redux store.

Actions creators are function that create and return action object. These functions are usually defined separately from components and can be imported and used to generate actions with specific types and payload.

Reducer interaction and working

Reducers are responsible for handling specific actions. When a action is dispatched, the relevant reducer with a matching action type is called to process the action and update the state accordingly.

Reducers are registered with the redux store. When a action is dispatched, the store passes the current state and the dispatched action to the appropriate reducer. The reducer then processes the action, updates the state based on the actions type and payload and return a new state. The store handles the process of passing the state and action to reducers.

Immutability libraries(spread operator or immutability-helper) are commonly used within reducers to create new copies of the state when making changes. This ensures the principal of redux such as immutability and a predictable state.

Store interaction and working

Components can read data from store using the useSelector hook. Component use selector to access specific parts of the state they need to render.

Middleware intercepts action before they react the reducers. Middleware can have access to the action, the current state and the dispatch function. Middleware can be used for tasks like handling asynchronous operations, logging and modify actions.

Reducer interact with the store indirectly by being registered with it. They don’t directly access the store but rather respond to dispatched actions by modifying the state passed to them.

useDispatch()

useDispatch hook is used in functional component to access the dispatch function provided by the redux store. The dispatch function is used to send actions to the store, which then triggers the reducers to update the state based on the dispatched actions.

useSelector()

useSelector hook is used in functional component to access specific parts of the state stored in the store. It allows components to “select” data from the store and automatically re render when that data changes.

Reduxjs

Redux is a predictable state container for javascript application. It is often used with libraries and frameworks like react to manage the state of a application in a predictable way. Redux follows the principal of unidirectional data flow and immutability, which makes is easier to understand and maintain the state of complex applications.

Unidirectional data flow:

In redux, the data flow within application follows a unidirectional pattern, meaning that data has a single direction of flow through the application or whole data in application follows the same lifecycle pattern.

Action reducer store view

Immutability

Immutability is the concept of not modifying data directly but instead creating a new copy with the desired changes.

In redux, the state is immutable, meaning that we don’t mutate the existing state but rather create a new object. the importance of immutability

Predictability

When we modify object or array in place, it can lead to unexpected side effects. Immutability ensures that we always know the state at a given point in time

Debugging

With immutability, we can easily track changes. If action is dispatched and the state is updated, we know that the old state remains unchanged.

Performance

Libraries and frameworks that implement optimization like memorization and changed detection rely on immutability to efficiently determine if updates have occurred.