**Redux middleware**

There are 3 core components in redux.

* Action
* Reducer
* Store

|  |  |
| --- | --- |
| Context API | Redux |
| Built in tool that ships with React | Additional installation required, driving up the final bundle size |
| Requires minimal setup | Require extensive setup to integrate it with a React application |
| Specifically designed for static data, that is not often refreshed or updated | It effectively manages both static and dynamic data, providing seamless control and updates throughout the application |
| Adding new contexts requires creation from scratch | Redux js allow for easy expansion by making it simple to add new data or action after the initial setup |
| Debugging can be hard in highly nested react component structure even with dev tool | Incredibly powerful redux dev tools to ease debugging |
| We can change state in it | The state is read only we cannot change them directly |
| It rerenders all components whenever there is any updates in the provider value prop | It only rerender the updated components |
| It is better to use with small application | It is perfect for larger application |
| It is easy to understand and requires less code | It is quite complex to understand |

**Comparing MVC, flux and redux**

**Architecture**

|  |  |  |
| --- | --- | --- |
| **MVC** | **Flux** | **Redux** |
| MVC model, view and controller | Application archietecture designed to build client side web apps | Open source javascript library. |

MVC architecture

MVC is well known 3 layer development architecture and it devides application into 3 components

Model it maintains the data and behavior of application

View Dispalys the model in the UI

Controller it servers as interface between view and model components

Middleware act between the action dispatch and the reducers. They can intercept actions and perform tasks such as logging, making asynchronous api calls, or modifying actions before they reach the reducers. Essentially they like filters.

Once the middleware processes the action, it is forwarded to the reducers. The reducers are pure function they take the current state and actions as the argument and return a new state based on the action. This is where the state transformation happens.

1. Redux thunk

Middleware that allow us to write action creators that return a function instead of action. This function can be used to delay the dispatch of action / to dispatch only if a certain condition is met. **Useful for handling simple asynchronous logic such as API calls.**

1. Redux saga

Middleware that allows us to handle complex asynchronous flows in our application using generators function. Sagas can be used to handle side effects, such as data fetching, and more complex control flows, like race condition and sequence actions**. Ideal for managing more complex asynchronous actions and side effects.**

1. Redux logger

Middleware that logs every action that is dispatched and the resulting new state,. It is useful for debugging and monitoring state changes. **Useful during development to track state changes and action dispatches for debugging purposes.**

1. Redux persist

Library that allows us to save and rehydrate the redux store in persistent storage like local storage. This means our redux state can persist across page reloads. **Useful for maintaining state between pages reloads / when a user returns to application after closing it.**

1. Redux loop

Redux loop that brings the concept of side effects into reducers, allowing us to return commands that are executed later and then dispatch actions based on their results. It helps in managing side effects in a functional programming style. Useful for structuring code in a way that keeps reducers pure and side effects explicitly defined.

**Redux thunk**

It is a middleware that allow us to write action creators that return function instead of action. This function receives the dispatch and getState function as argument, enabling us to perform asynchronous operations and dispatch actions based on the result of those operations.

Redux saga

It is also a middleware library used to handle side effects in redux applications, such as asynchronous operations, by using generator functions. It helps manage complex async flows more easily and can handle things like data fetching, user authentication, and other side effects.

**call** invokes a function and waits for it to complete

Invokes a function and waits for it to complete. It is used for making API calls / performing other side effects. **Pass the function to call along with any arguments it needs. Call is used within sagas to handle asynchronous tasks.**

import { call } from 'redux-saga/effects';

function\* fetchDataSaga() {

  try {

    const data = yield call(fetchDataFromApi);

    // handle data

  } catch (error) {

    // handle error

  }

}

**put** dispatches action to the store

Dispatches action to the redux store. This is used to send actions to reducers to update the stores state. **Pass the action object to put.**

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

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

function\* fetchDataSaga() {

  try {

    const data = yield call(fetchDataFromApi);

    yield put(fetchDataSuccess(data));

  } catch (error) {

    yield put(fetchDataFailure(error.message));

  }

}

**take** wait for a specific action to be dispatched

waits for a specific action to be dispatched and then continues execution. It is a blocking call, meaning the saga will pause until the action is dispatched. **Pass the action type / action creator to take**

import { take } from 'redux-saga/effects';

function\* watchAction() {

  yield take('ACTION\_TYPE');

  // action has been dispatched

}

**takeEvery** spawns a new saga for every action dispatched

listens for every action that matches the provided action type and spawns a new saga to handle each action. It is used for handling multiple action of the same type. **Pass the action type and the saga function to takeEvery.**

**takeLatest** only keeps the latest instance of a saga, cancelling previous instances

listens for the latest action that matches the provided action type and cancels any ongoing saga instance for same action type it is used when we only want the latest request to be handled, canceling previous ones. Pass the action type and the saga function to takeLatest.

**all** runs multiple saga in parallel

runs multiple sagas in parallel and waits for all of them to complete. Useful for combining multiple saga into one. Useful for combining multiple sagas into one.

**race** runs multiple sagas but only continues with the result of the first one to complete.

Runs multiple sagas in parallel but only continues with the result of the first saga that completes. It is used to handle scenarios where you need to race between multiple tasks. Pass object where each key is a saga to race.

**fork** starts a new saga in the background

starts a new saga in the background without blocking the execution of the parent saga. It is used to run sagas concurrently. **Pass the saga function to fork.**

**cancel** cancel saga task started with fork

cancels a saga task that was started by fork. It is used to stop the execution of a saga. Pass the task object returned by fork.

**select** accesses the state from within saga

accesses the redux store state from within a saga. It allows us to read the state and use it in our saga logic. Pass a selector function to select.

takeEvery and takeLatest

takeEvery listens for actions of a specific type. We pass the action type to takeEvery, so it only listens for actions of that type. It doesnt listen to all action / global action. for each action it will creates new instance of the saga function we provide. It means that if multiple actions of the same type are dispatched concurrently, multiple instances of the saga function will be run in parallel.

Each instance of the saga function will run independently. The saga function will receive the action object as argument can perform operation such as fetching data, processing the action, dispatching additional actions.

takeLatest handles only the most recent action that matches the specified action type. It cancels any ongoing saga instances for the same action type and starts a new one for the latest action.suitable for scenario where only the most recent action matter, such as making network request where only the latest result is relevant.

takeEvery for ACTION\_1 and ACTION\_2

takeEvery will handle every instance of the specified action types. If we have two separate ‘takeEvery’ watches for different actions, each will handle its respective action type independently.

takeEvery(‘ACTION\_1’, sagaForAction1) and takeEvery(‘ACTION\_2’, sagaForAction2) takeEvery will spawn a new instance of sagaForAction1 for each ACTION\_1 dispatch and a new instance of sagaForAction2 for each ACTION\_2 dispatched. Both sagas will run in parallel with respect to their respective action.

takeLatest for ACTION\_1 and ACTION\_2

similarly takeLatest will handle only the most recent for each specified action type, canceling any previous instances of the saga for same action type.

If we have takeLatest(‘ACTION\_1’,sagaForAction1) and takeLatest(‘ACTION\_2’,sagaForAction2) takeLatest will ensure that only the most recent ACTION\_1 is processed by sagaForAction1 and only the most recent ACTION\_2 is processed by sagaForAction2. Previous saga for ACTION\_1 / ACTION\_2 will be canceled if new action of the same type are dispatched.

Reason to cancel previous actions

Preventing redundant operation

In scenario where executing multiple instances of the same operation would be redundant / wasteful, canceling previous operations helps avoid unnecessary work. For example, if a user is continuously typing a search field and each keystroke triggers a search request, only the latest request might be relevant.

Avoiding race condition

When multiple asynchronous operation can interfere with each other, canceling previous operations helps prevent race conditions.. for instance, if we are making network requests where only the result of the latest request matters, cancelling previous requests ensures us only handle the most recent one.

Resource efficiency

Canceling previous operations can save resources network bandwidth, computational power by avoiding the execution of operations that are no longer needed.

Improving user experience

In cases where users might trigger multiple actions quickly,canceling previous actions helps ensure that the system in a coherent manner and does not show outdated or conflicting result.

Reason to don’t cancel previous actions

Independent operation

If each represents independent operation / task that should run to completion regardless of other actions, then there is no need to cancel previous actions. For example , processing multiple user-initiated actions, like uploading files, where each upload should complete independently.

Sequential operation

When operation need to be processed in sequence / their results should be considered cumulatively, canceling previous actions might be inappropriate. For example if you are handling a series of steps in a workflow, each step might depend on the previous ones.

Debugging and logging

Sometim we may want to keep track of all action and their outcomes for debugging / logging purposes. Canceling previous actions would discard useful information about what happened.

Search Autocomplete use takeLatest, when a user types into a search field and each keystroke trigger a search request, use takeLatest to ensure that only the latest search result is processes. Previous search requests are canceled to avoid processing outdated results.

File uploads use takeEvery, when users uploads multiple files,each file should be processed independently. Use takeEvery to handle each file upload request without canceling any ongoing uploads.