Redux itself is synchronous, so how the **async** operations like **network request** work with Redux? Here middlewares come handy. As discussed earlier, reducers are the place where all the execution logic is written. Reducer has nothing to do with who performs it, how much time it is taking or logging the state of the app before and after the action is dispatched.

In this case, Redux middleware function provides a medium to interact with dispatched action before they reach the reducer. Customized middleware functions can be created by writing high order functions (a function that returns another function), which wraps around some logic. Multiple middlewares can be combined together to add new functionality, and each middleware requires no knowledge of what came before and after. You can imagine middlewares somewhere between action dispatched and reducer.

Commonly, middlewares are used to deal with asynchronous actions in your app. Redux provides with API called applyMiddleware which allows us to use custom middleware as well as Redux middlewares like redux-thunk and redux-promise. It applies middlewares to store. The syntax of using applyMiddleware API is −

applyMiddleware(...middleware)

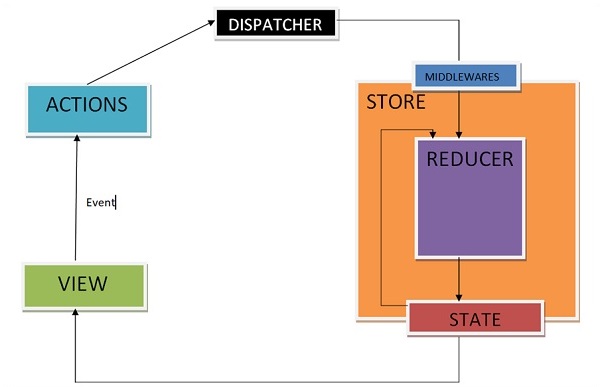
And this can be applied to store as follows −

import { createStore, applyMiddleware } from 'redux';

import thunk from 'redux-thunk';

import rootReducer from './reducers/index';

const store = createStore(rootReducer, applyMiddleware(thunk));



Middlewares will let you write an action dispatcher which returns a function instead of an action object. Example for the same is shown below −

function getUser() {

return function() {

return axios.get('/get\_user\_details');

};

}

Conditional dispatch can be written inside middleware. Each middleware receives store’s dispatch so that they can dispatch new action, and getState functions as arguments so that they can access the current state and return a function. Any return value from an inner function will be available as the value of dispatch function itself.

The following is the syntax of a middleware −

({ getState, dispatch }) => next => action

The getState function is useful to decide whether new data is to be fetched or cache result should get returned, depending upon the current state.

Let us see an example of a custom middleware logger function. It simply logs the action and new state.

import { createStore, applyMiddleware } from 'redux'

import userLogin from './reducers'

function logger({ getState }) {

return next => action => {

console.log(‘action’, action);

const returnVal = next(action);

console.log('state when action is dispatched', getState());

return returnVal;

}

}

Now apply the logger middleware to the store by writing the following line of code −

const store = createStore(userLogin , initialState=[ ] , applyMiddleware(logger));

Dispatch an action to check the action dispatched and new state using the below code −

store.dispatch({

type: 'ITEMS\_REQUEST',

isLoading: true

})

Another example of middleware where you can handle when to show or hide the loader is given below. This middleware shows the loader when you are requesting any resource and hides it when resource request has been completed.

import isPromise from 'is-promise';

function loaderHandler({ dispatch }) {

return next => action => {

if (isPromise(action)) {

dispatch({ type: 'SHOW\_LOADER' });

action

.then(() => dispatch({ type: 'HIDE\_LOADER' }))

.catch(() => dispatch({ type: 'HIDE\_LOADER' }));

}

return next(action);

};

}

const store = createStore(

userLogin , initialState = [ ] ,

applyMiddleware(loaderHandler)

);