Assignment-8

Q1: Why Middleware is important for React and Redux based applications?

Answer:

Middleware is some code you can put between the **framework** receiving a request, and the **framework** generating a response. **Redux Thunk middleware** allows you to write action creators that return a function instead of an action. The thunk can be used to delay the dispatch of an action, or to dispatch only for certain condition. The inner function receives the store methods dispatch and getState as parameters.

Redux Promise Middleware enables robust handling of async code in Redux. The middleware enables optimistic updates and dispatches pending, fulfilled and rejected actions. It can be combined with redux-thunk to chain async actions.

Redux Middleware provides a third-party extension point between dispatching an action, and the moment it reaches the reducer. People use Redux middleware for logging, crash reporting, talking to an asynchronous API, routing, and more.

Q2: What are the different approaches to use Logging in Middleware?

Answer:

[Attempt #1 : Logging Manually]
The most naïve solution is just to log the action and the next state yourself every time you call store.dispatch(action).

```
const action = addTodo('Use Redux')

console.log('dispatching', action)
store.dispatch(action)
console.log('next state', store.getState())
```

[Attempt #2 : Wrapping Dispatch]
You can extract logging into a function:

```
function dispatchAndLog(store, action) {
  console.log('dispatching', action)
  store.dispatch(action)
  console.log('next state', store.getState())
}
```

You can then use it everywhere instead of store.dispatch():

```
dispatchAndLog(store, addTodo('Use Redux'))
```

[Attempt #3: Monkeypatching Dispatch]

What if we just replace the dispatch function on the store instance? The Redux store is just a plain object with a few methods, and we're writing JavaScript, so we can just monkeypatch the dispatch implementation:

```
const next = store.dispatch
store.dispatch = function dispatchAndLog(action) {
  console.log('dispatching', action)
  let result = next(action)
  console.log('next state', store.getState())
  return result
}
```

This is already closer to what we want! No matter where we dispatch an action, it is guaranteed to be logged.

[Attempt #4: Hiding Monkeypatching]

Previously, our functions replaced store.dispatch. What if they *returned* the new dispatch function instead?

```
function logger(store) {
  const next = store.dispatch

  // Previously:
  // store.dispatch = function dispatchAndLog(action) {

  return function dispatchAndLog(action) {
    console.log('dispatching', action)
    let result = next(action)
    console.log('next state', store.getState())
    return result
  }
}
```

We could provide a helper inside Redux that would apply the actual monkeypatching as an implementation detail:

```
function applyMiddlewareByMonkeypatching(store, middlewares) {
  middlewares = middlewares.slice()
  middlewares.reverse()

  // Transform dispatch function with each middleware.
```

```
  middlewares.forEach(middleware => (store.dispatch = middleware(store)))
}
```

We could use it to apply multiple middleware like this:
applyMiddlewareByMonkeypatching(store, [logger, crashReporter])

However, it is still monkeypatching.
The fact that we hide it inside the library doesn't alter this fact.

[Attempt #5: Removing Monkeypatching]

The middleware could accept the next() dispatch function as a parameter instead of reading it from the store instance.

```
function logger(store) {
  return function wrapDispatchToAddLogging(next) {
    return function dispatchAndLog(action) {
      console.log('dispatching', action)
      let result = next(action)
      console.log('next state', store.getState())
      return result
    }
  }
}
```

ES6 arrow functions make this currying easier on eyes:

```
const logger = store => next => action => {
  console.log('dispatching', action)
  let result = next(action)
  console.log('next state', store.getState())
  return result
}

const crashReporter = store => next => action => {
  try {
    return next(action)
  } catch (err) {
    console.error('Caught an exception!', err)
    Raven.captureException(err, {
      extra: {
        action,
        state: store.getState()
      }
    })
    throw err
  }
```

```
}
```

This is exactly what Redux middleware looks like.

Now middleware takes the next() dispatch function, and returns a dispatch function, which in turn serves as next() to the middleware to the left, and so on. It's still useful to have access to some store methods like getState(), so storestays available as the top-level argument.

[Attempt #6: Naïvely Applying the Middleware]

Instead of applyMiddlewareByMonkeypatching(), we could write applyMiddleware() that first obtains the final, fully wrapped dispatch() function, and returns a copy of the store using it:

```
// Warning: Naïve implementation!
// That's *not* Redux API.
function applyMiddleware(store, middlewares) {
  middlewares = middlewares.slice()
  middlewares.reverse()
  let dispatch = store.dispatch
  middlewares.forEach(middleware => (dispatch = middleware(store)(dispatch)))
  return Object.assign({}, store, { dispatch })
}
```

The implementation of applyMiddleware() that ships with Redux is similar, but **different in three important aspects**:

- It only exposes a subset of the store API to the middleware: dispatch(action) and getState().
- It does a bit of trickery to make sure that if you call store.dispatch(action) from your middleware instead of next(action), the action will actually travel the whole middleware chain again, including the current middleware. This is useful for asynchronous middleware, as we have seen previously. There is one caveat when calling dispatch during setup, described below.
- To ensure that you may only apply middleware once, it operates on createStore() rather than on store itself. Instead of (store, middlewares) => store, its signature is (...middlewares) => (createStore) => createStore.

Because it is cumbersome to apply functions to createStore() before using it, createStore() accepts an optional last argument to specify such functions.

Caveat: Dispatching During Setup
While applyMiddleware executes and sets up your middleware, the store.dispatch function will point to the vanilla version provided by createStore. Dispatching would result in no other middleware being applied. If you are expecting an interaction with another middleware during setup, you will probably be disappointed. Because of this unexpected behavior, applyMiddleware will throw an error if you try to dispatch an action before the set up completes. Instead, you should either communicate directly with that other middleware via a common object (for an API-calling middleware, this may be your API client object) or waiting until after the middleware is constructed with a callback.

[The Final Approach]

Given this middleware we just wrote:

```
const logger = store => next => action => {
  console.log('dispatching', action)
  let result = next(action)
  console.log('next state', store.getState())
  return result
}

const crashReporter = store => next => action => {
  try {
    return next(action)
  } catch (err) {
    console.error('Caught an exception!', err)
    Raven.captureException(err, {
      extra: {
        action,
        state: store.getState()
      }
    })
    throw err
  }
}
```

Here's how to apply it to a Redux store:

```
import { createStore, combineReducers, applyMiddleware } from 'redux'

const todoApp = combineReducers(reducers)
const store = createStore(
  todoApp,
  // applyMiddleware() tells createStore() how to handle middleware
  applyMiddleware(logger, crashReporter)
)
```

That's it! Now any actions dispatched to the store instance will flow through logger and crashReporter:

```
// Will flow through both logger and crashReporter middleware!

store.dispatch(addTodo('Use Redux'))
```

Reference : https://redux.js.org/advanced/middleware