Redux 3.2_CW Exercises

ex01: update likes

understanding

When updating the state there are few new and old things which we need to understand:

1. Reducers are objects now. You see that we are not using `switch` anymore. This is much better as it keeps the code clean and also faster than switch.
    1. I covered `switch` because most of the examples around the internet was that and it's beginner friendly. But now you can move even your reducers in `useReducer` to objects if you want. More on that later.
2. Actions are generated automatically and are functions. In typical redux also there was the convention of creating functions that would return an object `{ type: string, payload?: any}`. Now, with the toolkit, you don't need to write these functions by hand anymore. Big relief otherwise there was a lot of boilerplate earlier!
3. Immer. Notice that you don't need to do immutable updates on state anymore. Toolkit comes with a library immer which makes doing immutable updates a breeze. We all know how hard it is to do updates on nested objects.

Once again, you can use immer in your useReducer as well.

However, immer with toolkit has it's gotchas. Even a `console.log()` doesn't work exactly as expected. Therefore, definitely read this when you're working on the app: https://redux-toolkit.js.org/usage/immer-reducers

4. Dispatch.
    1. To dispatch an action you need `useDispatch()` in the component.
    2. Notice that the action function is being called inside the `dispatch()` call.

challenge

Create a button against the likes number. On click of this button, the number of likes should increase.

ex01.1: implementing the `likeButtonPressed` action

challenge

In your Redux `**postSlice.js**`, implement the `**likeButtonPressed**` action. This action should increase the number of likes for a specific post when the "Like" button is pressed.

1. Inside the `reducers` object of the `createSlice` function, add a new reducer named `likeButtonPressed`. This reducer takes two arguments: `state` and `action`.
2. Within the `likeButtonPressed` reducer, you need to find the index of the post you want to update based on the `postID` provided in the `action.payload`. To do this, use the `findIndex` method on the `state.posts` array.
3. Check if the `postIndex` is not equal to -1, indicating that the post with the given `postID` was found in the `state.posts` array.
4. If the `postIndex` is not -1, increment the `likes` count of the corresponding post in the `state.posts` array by 1. You can access the post using `state.posts[postIndex]` and update its `likes` property.
5. Export the `likeButtonPressed` action using the destructuring assignment.

solution

```
import { createSlice } from '@reduxjs/toolkit'

export const postSlice = createSlice({
  name: 'posts',
  initialState: {},
  reducers: {
    likeButtonPressed: (state, action) => {
      const postIndex = state.posts.findIndex(
        (post) => post.postID === action.payload,
      )

      if (postIndex !== -1) {
        state.posts[postIndex].likes += 1
      }
    },
  },
})

export const { likeButtonPressed } = postSlice.actions

export default postSlice.reducer
```

COPY

ex01.2: using `likeButtonPressed` in the `Posts` component

challenge

In your `**Posts.js**` component, use the `**likeButtonPressed**` action from Redux to handle the "Like" button click for each post. Dispatch this action when the button is clicked to increase the number of likes for the corresponding post.

1. Obtain the `dispatch` function from `**useDispatch**` hook from the `react-redux` library. This function is used to dispatch actions to the Redux store.
2. In the `onClick` handler of the "Like" button, call the `dispatch` function and pass in the `likeButtonPressed` action with the `post.postID` as the payload. This action will be dispatched when the "Like" button is clicked for a specific post.

solution

```
import React from 'react'
import { useSelector, useDispatch } from 'react-redux'
import { likeButtonPressed } from './postSlice'

export default function Posts() {
  const posts = useSelector((state) => {
    console.log({ state })
    return state.posts
  })
  const dispatch = useDispatch()

  return (
    <div>
      <div>
        {posts.posts.map((post) => (
          <article key={post.postID} className='post'>
            <div className='caption'> {post.caption} </div>
            <button
              className='likes'
              onClick={() => dispatch(likeButtonPressed(post.postID))}
            >
              {post.likes} likes{' '}
            </button>
          </article>
        ))}
      </div>
    </div>
  )
}
```

COPY

entire solution

https://codesandbox.io/s/react-redux-ex02-sol-9z3u3

ex02: load initial data from the server

understanding

- thunk https://github.com/reduxjs/redux-thunk#why-do-i-need-this
  - middleware: Exactly similar to how it works in the express. Gives you a superpower in the dispatch lifecycle.
  - thunk: Essentially a function which returns a function. More details are for purists.
- createAsyncThunk https://redux-toolkit.js.org/api/createAsyncThunk ←- documentation to look for challenge Makes your life much easier by abstracting the same work of dispatching
  - `loading`,
  - `fulfilled`, and
  - `rejected` actions in every async request. Note, you don't need to put all three in every API request. Some actions can just have `fulfilled` and `error` states.
- status: As discussed in previous classes as well, status shouldn't be boolean but an enum of different strings. With redux, we are storing all the data, ie posts in the global state directly. In this scenario, redux also acts as a global cache. You don't need to load posts every time you're on that route say `/posts` from `/user` route right? To do that you need to dispatch the `fetchPosts()` action only when the status is `idle`. `js { status: 'idle' | 'loading' | 'succeeded' | 'failed', error: string | null }`

challenge

server: https://social-media-server.tanaypratap.repl.co/posts

Use a server or a fakeApi to make the initial data asynchronous.

Now, load that data into the app.

Show loading and fulfilled states.

Bonus: Show error as well.

ex02.1: writing `fetchPosts` async thunk

challenge

In your Redux slice file (`postSlice.js`), write the `fetchPosts` async thunk using `createAsyncThunk` to fetch data from the specified server URL. Ensure that it handles loading, fulfilled, and error states.

solution

```
import { createAsyncThunk, createSlice } from '@reduxjs/toolkit'
import axios from 'axios'

export const fetchPosts = createAsyncThunk('posts/fetchPosts', async () => {
  const response = await axios.get(
    'https://social-media-server.tanaypratap.repl.co/posts',
  )
  return response.data
})
```

COPY

ex02.2: updating `initialState`

challenge

In your Redux slice file (`postSlice.js`), write the `initialState` for your `posts` slice with status and error.

solution

```
import { createAsyncThunk, createSlice } from '@reduxjs/toolkit'
import axios from 'axios'

const initialState = {
  status: 'idle',
  error: null,
  posts: [],
}

const postSlice = createSlice({
  name: 'posts',
  initialState,
  reducers: {},
})
```

COPY

ex02.3: writing `extraReducers` for `fetchPosts`

challenge

In your Redux slice file (`postSlice.js`), write the `extraReducers` to handle the `fetchPosts` async thunk actions (`pending`, `fulfilled`, `rejected`).

solution

```
import { createAsyncThunk, createSlice } from '@reduxjs/toolkit'
import axios from 'axios'
```

```
// Define the initial state and reducers (from previous exercises)

const postSlice = createSlice({
  name: 'posts',
  // ... (from previous exercises)
  extraReducers: {
    [fetchPosts.pending]: (state) => {
      state.status = 'loading'
    },
    [fetchPosts.fulfilled]: (state, action) => {
      state.status = 'success'
      state.posts = action.payload.posts
    },
    [fetchPosts.rejected]: (state, action) => {
      state.status = 'error'
      console.log(action.error.message)
      state.error = action.error.message
    },
  },
})
```

COPY

ex02.4: using `fetchPosts` in `Posts` component

challenge

In your `Posts.js` component, use the `fetchPosts` async thunk to fetch data when the status is "idle." Display loading, error, and post data based on the status.

solution

```
import React, { useEffect } from 'react'
import { useDispatch, useSelector } from 'react-redux'
import { fetchPosts, incrementLike } from './postSlice'

export function Posts() {
  const { posts, status, error } = useSelector((state) => state.posts)
  const dispatch = useDispatch()

  useEffect(() => {
    if (status === 'idle') {
      dispatch(fetchPosts())
    }
  }, [status, dispatch])

  return (
    <div>
      {status === 'loading' && <p>Loading...</p>}
      {status === 'error' && <p>{error}</p>}
      <div>
        {posts.map((post) => (
          <div key={post.postID}>
            <p>{post.caption}</p>
```

```
            <button onClick={() => dispatch(incrementLike(post.postID))}>
              {post.likes}
            </button>
          </div>
        ))}
      </div>
    </div>
  )
}
```

COPY

entire solution

https://codesandbox.io/s/react-redux-ex03-prac-01-wg1rv