codecademy

Async Actions with Middleware and Thunks

Thunks

A *thunk* is a function used to delay a computation until it is needed by an application. The term thunk comes from a play on the word “think” but in the past tense. In JavaScript, functions are thunks since they hold a computation and they can be executed at any time or passed to another function to be executed at any time. A common practice is for thunks to be returned by a higher-order function. The returned thunk contains the process that is to be delayed until needed.

```
const alarmOne = () => {
  console.log("Wake Up!!!");
};
alarmOne(); // "Wake Up!!!"

const getAlarmThunk = () => {
    return () => {
    console.log("Wake Up!!!");
  }
};
const alarmTwo = getAlarmThunk();
alarmTwo(); // "Wake Up!!!"
```

Thunks in Redux

In Redux thunks can be used to hold asynchronous logic that interacts with the Redux store. When thunks are dispatched to the store the enclosed asynchronous computations are evaluated before making it to the Redux store. The arguments passed to thunks are the Redux store methods `dispatch` and `getState`. This allows actions to be dispatched or for the state to be referenced within the containing logic.
Other benefits of thunks are:

- Creating abstract logic that can interact with any Redux store
- Move complex logic out of components

```
import { fetchTodos } from '../actions';

const fetchTodosThunk = (
  dispatch,
  getState
) => {
  setTimeout(
    dispatch(fetchTodos()),
    5000);
};
```

Middleware In Redux

Redux *middleware* extends the store’s abilities and lets you write asynchronous logic that can interact with the store. Middleware is added to the store either through `createStore()` or `configureStore()`.
The `redux-thunk` package is a popular tool when using middleware in a Redux application.

Redux Thunk Middleware

The `redux-thunk` middleware package 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 if a certain condition is met.

```
import { fetchTodos } from '../actions';

const fetchTodosLater = () => {
  return (dispatch, getState) => {
    setTimeout(
    dispatch(fetchTodos()),
    5000);
    }
};
/*
redux-thunk allows the returned
thunk to be dispatched
*/
store.dispatch(fetchTodosLater());
```

The `redux-thunk` Package

The `redux-thunk` package is included in the Redux Toolkit (`@reduxjs/redux-toolkit`) package. To install `@reduxjs/redux-toolkit` or the standalone `redux-thunk` package use `npm` .
The `redux-thunk` middleware allows for asynchronous logic when interacting with the Redux store.

```
npm install @reduxjs/redux-toolkit

npm install redux-thunk
```

`createAsyncThunk()

`createAsyncThunk()` accepts a Redux action type string and a callback function that should return a promise. It generates promise lifecycle action types based on the action type prefix that you pass in, and returns a thunk action creator that will run the promise callback and dispatch the lifecycle actions based on the returned promise.
The callback function takes a user-defined data argument and a thunkAPI object argument. The data argument is originally sent as an argument to the thunk action creator where an object can be used if multiple points of data are necessary. The thunkAPI object contains the usual thunk arguments such as dispatch and getState.

```
import { createAsyncThunk } from
'@reduxjs/toolkit'
import { userAPI } from './userAPI'

const fetchUser = createAsyncThunk(
  'users/fetchByIdStatus',
  async (user, thunkAPI) => {
    const response = await
userAPI.fetchById(user.id)
    return response.data
  }
)

const user = {username: "coder123", id:
3};
store.dispatch(fetchUser(user))
```

`extraReducers` Property

The object passed to `createSlice()` may contain a fourth property, `extraReducers`, which allows `createSlice()` to respond to other action types besides the types it has generated. This is useful when handling asynchronous logic using thunks.

The logic within `extraReducers` that acts on the slice of `state` can safely use mutatable updates because it uses Immer internally.

```
import { createSlice, createAsyncThunk
} from '@reduxjs/toolkit';
import { client } from '../api';

const initialState = {
  todos: [],
  status: 'idle'
};

export const fetchTodos
= createAsyncThunk('todos/fetchTodos',
async () => {
  const response = await
client.get('/todosApi/todos');
  return response.todos;
});

const todosSlice = createSlice({
  name: 'todos',
  initialState,
  reducers: {
    addTodo: (state, action) => {
      state.todos.push(action.payload);
    }
  },
  extraReducers: {
    [fetchTodos.pending]: (state, action)
=> {
      state.status = 'loading';
    },
    [fetchTodos.fulfilled]: (state,
action) => {
      state.status = 'succeeded';
      state.todos
= state.todos.concat(action.payload);
    }
  }
});
```