Asynchronous Redux

Asynchronous Redux refers to handling asynchronous operations, such as API calls or timeouts, within a Redux application. Redux itself is synchronous, so managing asynchronous actions requires additional middleware to handle side effects.

Why Asynchronous Redux?

Redux is a predictable state container for JavaScript apps, and it operates synchronously. This means that actions are dispatched and reducers handle these actions immediately in a synchronous manner. However, many real-world applications require dealing with asynchronous operations, such as:

- **API Requests:** Fetching data from a server.
- **Timeouts:** Delaying actions to simulate loading states.
- **WebSockets:** Handling real-time updates.

Without a proper mechanism to handle these asynchronous operations, state management could become difficult and less predictable.

Key Concepts

- **Asynchronous Actions:** These are actions that involve operations that don't happen instantly, like fetching data from a server or performing time-based tasks.
- **Middleware:** Middleware in Redux allows you to extend Redux's capabilities and handle side effects. Middleware sits between the dispatch of an action and the moment it reaches the reducer. For asynchronous operations, middleware is essential to manage actions that occur asynchronously.

Common Middleware for Asynchronous Actions

- **Redux Thunk:** This middleware allows action creators to return a function instead of an action object. This function can perform asynchronous operations and dispatch actions based on the result.
- **Redux Saga:** This middleware uses generator functions to handle side effects. It provides a more complex but powerful way to manage complex asynchronous workflows and side effects.

Redux Thunk

In Redux, a "thunk" is a middleware function that allows you to write action creators that return functions instead of plain action objects. This can be particularly useful for handling asynchronous logic, such as making API calls or performing other side effects.

What is a Thunk?

- A thunk is a function that encapsulates some logic that you want to execute later.
- In the context of Redux, a thunk is a function that allows you to delay the dispatch of an action or perform asynchronous operations before dispatching an action.

Why Use Thunks?

- **Handling Asynchronous Operations:** Redux itself is synchronous, meaning actions and reducers run synchronously. Thunks allow you to handle asynchronous logic (e.g., API requests) within your action creators.
- **Conditionally Dispatching Actions:** Thunks enable you to conditionally dispatch actions based on some logic or state.
- **Encapsulating Complex Logic:** You can use thunks to encapsulate complex logic that involves multiple steps or side effects.

How It Works

- When an action creator returns a function, Redux Thunk intercepts it.
- The function receives dispatch and getState as arguments, allowing it to dispatch additional actions or access the current state.
- After performing the asynchronous task, the function can dispatch actions to update the state.

Example Flow

- **Dispatch Action:** An action creator returns a function.
- **Perform Async Task:** The function performs the asynchronous operation (e.g., API call).
- **Dispatch Success/Failure:** Based on the result, the function dispatches success or failure actions.

Example : simple example to illustrate how thunks work

1. First, install parcel, redux and redux-thunk

```
npm install parcel redux redux-thunk
```

2. Create a Redux Store with Thunk Middleware

store.js

```
import { createStore, applyMiddleware } from 'redux';
import {thunk} from 'redux-thunk';
import rootReducer from './reducers.js';

const store = createStore(
  rootReducer,
  applyMiddleware(thunk) // Apply thunk middleware
);

export default store;
```

3. Define an Action Creator with a Thunk

actions.js

```
// Action Types
const FETCH_DATA_REQUEST = 'FETCH_DATA_REQUEST';
const FETCH_DATA_SUCCESS = 'FETCH_DATA_SUCCESS';
const FETCH_DATA_FAILURE = 'FETCH_DATA_FAILURE';

// Action Creators
const fetchDataRequest = () => ({ type: FETCH_DATA_REQUEST });
const fetchDataSuccess = (data) => ({ type: FETCH_DATA_SUCCESS, payload: data
});
const fetchDataFailure = (error) => ({ type: FETCH_DATA_FAILURE, payload: error
});

// Thunk Action Creator
export const fetchData = () => {
  return async (dispatch) => {
    dispatch(fetchDataRequest());
    try {
      const response = await fetch('https://jsonplaceholder.typicode.com/posts');
      if (!response.ok) {
        throw new Error('Network response was not ok');
      }
      const data = await response.json();
      dispatch(fetchDataSuccess(data));
    } catch (error) {
      dispatch(fetchDataFailure(error.toString()));
    }
  };
};
```

4. Handle Thunk Actions in Reducers
   reducers.js

```
const initialState = {
  loading: false,
  data: [],
  error: null
};

const dataReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'FETCH_DATA_REQUEST':
      return { ...state, loading: true, error: null };
    case 'FETCH_DATA_SUCCESS':
      return { ...state, loading: false, data: action.payload };
    case 'FETCH_DATA_FAILURE':
      return { ...state, loading: false, error: action.payload };
    default:
      return state;
  }
};

export default dataReducer;
```

5. Dispatch the Thunk
   main.js

```
import store from './store.js';
import { fetchData } from './actions.js';

// Dispatch the thunk to fetch data
store.dispatch(fetchData());

// Subscribe to store updates and log the state
store.subscribe(() => {
  const state = store.getState();
  console.log('Current state:', state);
});
```

6. Create index.html to sets up a simple HTML page.
   index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Thunk Example</title>
</head>
<body>
  <h1>Check the console for fetched data</h1>

  <script type="module" src="main.js"></script>
</body>
</html>
```

7. Add Build Scripts by updating your package.json with

```
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "parcel build index.html",
    "start": "parcel index.html"
  }
```

createAsyncThunk

- createAsyncThunk is a utility function provided by Redux Toolkit to simplify handling asynchronous operations in Redux.
- It allows you to create a thunk that dispatches actions based on the lifecycle of an asynchronous operation (e.g., a network request). This function is particularly useful for handling side effects such as data fetching.
- It automatically generates action creators and action types for the pending, fulfilled, and rejected states of an async operation.

Why Use createAsyncThunk?

Managing async actions directly with Redux can be difficult and complex, involving a lot of boilerplate code for dispatching actions and handling the different states of the request (e.g., loading, success, error). createAsyncThunk simplifies this process by handling these tasks for you.

How Does createAsyncThunk Work?

- **Define the Async Operation:** You pass a function that performs the asynchronous operation. This function will be called with an argument (if any) and should return a promise.
- **Lifecycle Actions:** createAsyncThunk automatically generates three action types and corresponding action creators:
  - **Pending:** Dispatched when the async operation starts.
  - **Fulfilled:** Dispatched when the async operation succeeds.
  - **Rejected:** Dispatched when the async operation fails.
- **Reducer Handling:** You handle these actions in the extraReducers field of a slice to update the state accordingly.

Syntax

createAsyncThunk(typePrefix, payloadCreator, options)

where,

- typePrefix: A string used as a prefix for the action types (e.g., 'posts/fetchPosts').
- payloadCreator: A function that returns a promise. It can be asynchronous and takes a single argument (the payload).
- options (optional): Additional options to configure the thunk (e.g., condition, dispatch).

Example: Redux Toolkit to fetch data from the JSONPlaceholder API and log the data to the console.

1. First, install parcel and redux-toolkit

```
npm install parcel @reduxjs/toolkit
```

2. Create an HTML file with a basic setup to load the JavaScript.

   index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Fetch Data with Redux Toolkit</title>
</head>
<body>
  <h1>Check the console for fetched data</h1>

  <script type="module" src="main.js"></script>
</body>
</html>
```

3. Define a Redux slice for fetching and storing posts.

   postsSlice.js

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

// Async thunk for fetching posts
export const fetchPosts = createAsyncThunk('posts/fetchPosts', async () =>
{
  const response = await
fetch('https://jsonplaceholder.typicode.com/posts');
  if (!response.ok) {
    throw new Error('Network response was not ok');
  }
  return response.json();
});
```

```
const postsSlice = createSlice({
  name: 'posts',
  initialState: {
                        posts: [],
                        status: 'idle',
                        error: null
  },
  reducers: {},
  extraReducers: (builder) => {
    builder
      .addCase(fetchPosts.pending, (state) => {
        state.status = 'loading';
      })
      .addCase(fetchPosts.fulfilled, (state, action) => {
        state.status = 'succeeded';
        state.posts = action.payload;
      })
      .addCase(fetchPosts.rejected, (state, action) => {
        state.status = 'failed';
        state.error = action.error.message;
      });
  }
});

export const selectPosts = (state) => state.posts.posts;
export const postsReducer = postsSlice.reducer;
```

4. Set up the Redux store.
   store.js

```
import { configureStore } from '@reduxjs/toolkit';
import { postsReducer } from './postsSlice.js';

export const store = configureStore({
  reducer: {
    posts: postsReducer
  }
});
```

5. Fetch data from JSONPlaceholder and log it to the console.
   main.js
   ```
   import { store } from './store.js';
   import { fetchPosts, selectPosts } from './postsSlice.js';

   // Dispatch the fetchPosts action
   store.dispatch(fetchPosts()).then(() => {
     // Select the posts from the store and log them to the console
     const posts = selectPosts(store.getState());
     console.log('Fetched posts:', posts);
   }).catch((error) => {
     console.error('Failed to fetch posts:', error);
   });
   ```

6. Add Build Scripts by updating your package.json to include:
   ```
   "scripts": {
       "test": "echo \"Error: no test specified\" && exit 1",
       "build": "parcel build index.html",
       "start": "parcel index.html"
     }
   ```