Redux Essentials, Part 5: Async Logic and Data Fetching

WHAT YOU'LL LEARN

- How to use the Redux "thunk" middleware for async logic
- Patterns for handling async request state
- How to use the Redux Toolkit `createAsyncThunk` API to simplify async calls

PREREQUISITES

- Familiarity with using AJAX requests to fetch and update data from a server

Introduction

In Part 4: Using Redux Data, we saw how to use multiple pieces of data from the Redux store inside of React components, customize the contents of action objects before they're dispatched, and handle more complex update logic in our reducers.

So far, all the data we've worked with has been directly inside of our React client application. However, most real applications need to work with data from a server, by making HTTP API calls to fetch and save items.

In this section, we'll convert our social media app to fetch the posts and users data from an API, and add new posts by saving them to the API.

TIP

Redux Toolkit includes the **RTK Query data fetching and caching API**. RTK Query is a purpose built data fetching and caching solution for Redux apps, and **can eliminate the need to write *any* thunks or reducers to manage data fetching**. We specifically teach RTK Query as the default approach for data fetching, and RTK Query is built on the same patterns shown in this page.

We'll cover how to use RTK Query starting in Part 7: RTK Query Basics.

Example REST API and Client

To keep the example project isolated but realistic, the initial project setup already includes a fake in-memory REST API for our data (configured using the Mock Service Worker mock API tool). The API uses `/fakeApi` as the base URL for the endpoints, and supports the typical `GET/POST/PUT/DELETE` HTTP methods for `/fakeApi/posts`, `/fakeApi/users`, and `fakeApi/notifications`. It's defined in `src/api/server.js`.

The project also includes a small HTTP API client object that exposes `client.get()` and `client.post()` methods, similar to popular HTTP libraries like `axios`. It's defined in `src/api/client.js`.

We'll use the `client` object to make HTTP calls to our in-memory fake REST API for this section.

Also, the mock server has been set up to reuse the same random seed each time the page is loaded, so that it will generate the same list of fake users and fake posts. If you want to reset that, delete the `'randomTimestampSeed'` value in your browser's Local Storage and reload the page, or you can turn that off by editing `src/api/server.js` and setting `useSeededRNG` to `false`.

> **INFO**
>
> As a reminder, the code examples focus on the key concepts and changes for each section. See the CodeSandbox projects and the `tutorial-steps` branch in the project repo for the complete changes in the application.

Thunks and Async Logic

Using Middleware to Enable Async Logic

By itself, a Redux store doesn't know anything about async logic. It only knows how to synchronously dispatch actions, update the state by calling the root reducer function, and notify the UI that something has changed. Any asynchronicity has to happen outside the store.

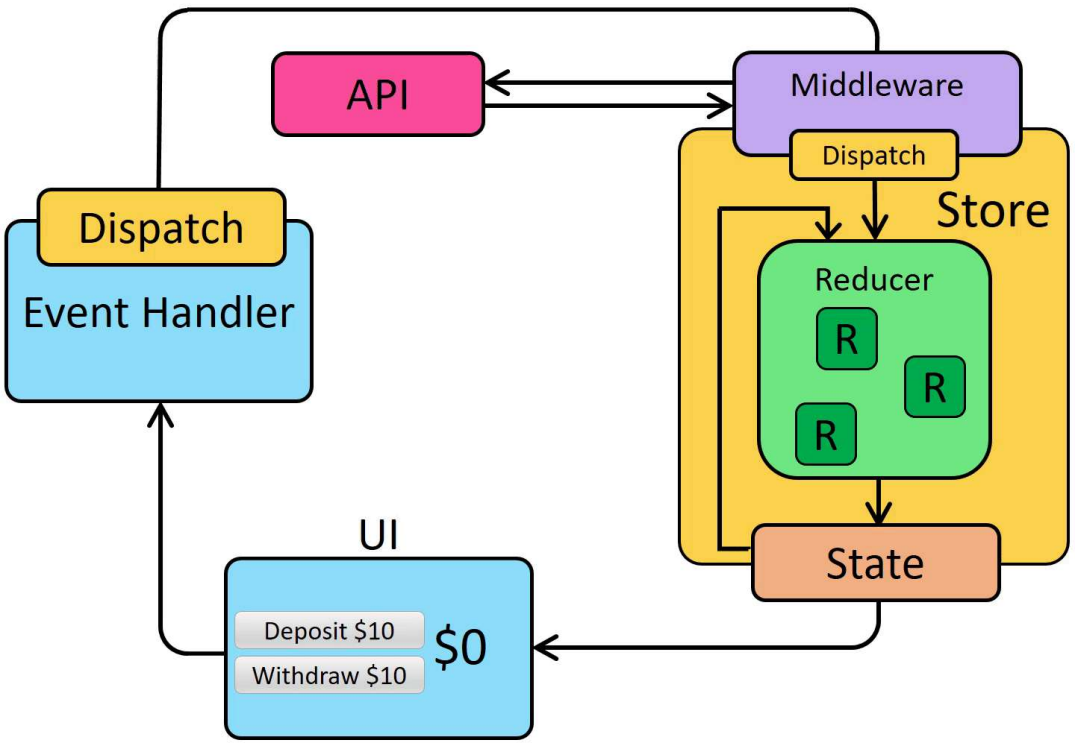
But, what if you want to have async logic interact with the store by dispatching or checking the current store state? That's where Redux middleware come in. They extend the store, and allow you to:

- Execute extra logic when any action is dispatched (such as logging the action and state)
- Pause, modify, delay, replace, or halt dispatched actions
- Write extra code that has access to `dispatch` and `getState`
- Teach `dispatch` how to accept other values besides plain action objects, such as functions and promises, by intercepting them and dispatching real action objects instead

The most common reason to use middleware is to allow different kinds of async logic to interact with the store. This allows you to write code that can dispatch actions and check the store state, while keeping that logic separate from your UI.

There are many kinds of async middleware for Redux, and each lets you write your logic using different syntax. The most common async middleware is `redux-thunk`, which lets you write plain functions that may contain async logic directly. Redux Toolkit's `configureStore` function automatically sets up the thunk middleware by default, and we recommend using thunks as a standard approach for writing async logic with Redux.

Earlier, we saw what the synchronous data flow for Redux looks like. When we introduce asynchronous logic, we add an extra step where middleware can run logic like AJAX requests, then dispatch actions. That makes the async data flow look like this:



Thunk Functions

Once the thunk middleware has been added to the Redux store, it allows you to pass *thunk functions* directly to `store.dispatch`. A thunk function will always be called with `(dispatch, getState)` as its arguments, and you can use them inside the thunk as needed.

Thunks typically dispatch plain actions using action creators, like `dispatch(increment())`:

```
const store = configureStore({ reducer: counterReducer })

const exampleThunkFunction = (dispatch, getState) => {
  const stateBefore = getState()
  console.log(`Counter before: ${stateBefore.counter}`)
  dispatch(increment())
  const stateAfter = getState()
  console.log(`Counter after: ${stateAfter.counter}`)
}

store.dispatch(exampleThunkFunction)
```

For consistency with dispatching normal action objects, we typically write these as *thunk action creators*, which return the thunk function. These action creators can take arguments that can be used inside the thunk.

```
const logAndAdd = amount => {
  return (dispatch, getState) => {
    const stateBefore = getState()
    console.log(`Counter before: ${stateBefore.counter}`)
    dispatch(incrementByAmount(amount))
    const stateAfter = getState()
    console.log(`Counter after: ${stateAfter.counter}`)
  }
}

store.dispatch(logAndAdd(5))
```

Thunks are typically written in "slice" files. `createSlice` itself does not have any special support for defining thunks, so you should write them as separate functions in the same slice file. That way, they have access to the plain action creators for that slice, and it's easy to find where the thunk lives.

> **INFO**
>
> The word "thunk" is a programming term that means "a piece of code that does some delayed work". For more details on how to use thunks,

see the thunk usage guide page:

- Using Redux: Writing Logic with Thunks

as well as these posts:

- What the heck is a thunk?
- Thunks in Redux: the basics

Writing Async Thunks

Thunks may have async logic inside of them, such as `setTimeout`, `Promise`s, and `async/await`. This makes them a good place to put AJAX calls to a server API.

Data fetching logic for Redux typically follows a predictable pattern:

- A "start" action is dispatched before the request, to indicate that the request is in progress. This may be used to track loading state to allow skipping duplicate requests or show loading indicators in the UI.
- The async request is made
- Depending on the request result, the async logic dispatches either a "success" action containing the result data, or a "failure" action containing error details. The reducer logic clears the loading state in both cases, and either processes the result data from the success case, or stores the error value for potential display.

These steps are not *required*, but are commonly used. (If all you care about is a successful result, you can just dispatch a single "success" action when the request finishes, and skip the "start" and "failure" actions.)

Redux Toolkit provides a `createAsyncThunk` API to implement the creation and dispatching of these actions, and we'll look at how to use it shortly.

▶ **Detailed Explanation: Dispatching Request Status Actions in Thunks**

Loading Posts

So far, our `postsSlice` has used some hardcoded sample data as its initial state. We're going to switch that to start with an empty array of posts instead, and then fetch a list of posts from the server.

In order to do that, we're going to have to change the structure of the state in our `postsSlice`, so that we can keep track of the current state of the API request.

Extracting Posts Selectors

Right now, the `postsSlice` state is a single array of `posts`. We need to change that to be an object that has the `posts` array, plus the loading state fields.

Meanwhile, the UI components like `<PostsList>` are trying to read posts from `state.posts` in their `useSelector` hooks, assuming that field is an array. We need to change those locations also to match the new data.

It would be nice if we didn't have to keep rewriting our components every time we made a change to the data format in our reducers. One way to avoid this is to define reusable selector functions in the slice files, and have

the components use those selectors to extract the data they need instead of repeating the selector logic in each component. That way, if we do change our state structure again, we only need to update the code in the slice file.

The `<PostsList>` component needs to read a list of all the posts, and the `<SinglePostPage>` and `<EditPostForm>` components need to look up a single post by its ID. Let's export two small selector functions from `postsSlice.js` to cover those cases:

features/posts/postsSlice.js

```
const postsSlice = createSlice(/* omit slice code*/)

export const { postAdded, postUpdated, reactionAdded } =
postsSlice.actions

export default postsSlice.reducer

export const selectAllPosts = state => state.posts

export const selectPostById = (state, postId) =>
  state.posts.find(post => post.id === postId)
```

Note that the `state` parameter for these selector functions is the root Redux state object, as it was for the inlined anonymous selectors we wrote directly inside of `useSelector`.

We can then use them in the components:

features/posts/PostsList.js

```
// omit imports
import { selectAllPosts } from './postsSlice'

export const PostsList = () => {
  const posts = useSelector(selectAllPosts)
  // omit component contents
}
```

features/posts/SinglePostPage.js

```
// omit imports
import { selectPostById } from './postsSlice'

export const SinglePostPage = ({ match }) => {
  const { postId } = match.params

  const post = useSelector(state => selectPostById(state,
postId))
  // omit component logic
}
```

features/posts/EditPostForm.js

```
// omit imports
import { postUpdated, selectPostById } from './postsSlice'

export const EditPostForm = ({ match }) => {
  const { postId } = match.params

  const post = useSelector(state => selectPostById(state,
postId))
  // omit component logic
}
```

It's often a good idea to encapsulate data lookups by writing reusable selectors. You can also create "memoized" selectors that can help improve

performance, which we'll look at in a later part of this tutorial.

But, like any abstraction, it's not something you should do *all* the time, everywhere. Writing selectors means more code to understand and maintain. **Don't feel like you need to write selectors for every single field of your state**. Try starting without any selectors, and add some later when you find yourself looking up the same values in many parts of your application code.

Loading State for Requests

When we make an API call, we can view its progress as a small state machine that can be in one of four possible states:

- The request hasn't started yet
- The request is in progress
- The request succeeded, and we now have the data we need
- The request failed, and there's probably an error message

We *could* track that information using some booleans, like `isLoading: true`, but it's better to track these states as a single enum value. A good pattern for this is to have a state section that looks like this (using TypeScript type notation):

```
{
  // Multiple possible status enum values
  status: 'idle' | 'loading' | 'succeeded' | 'failed',
  error: string | null
}
```

These fields would exist alongside whatever actual data is being stored. These specific string state names aren't required - feel free to use other names if you want, like `'pending'` instead of `'loading'`, or `'complete'` instead of `'succeeded'`.

We can use this information to decide what to show in our UI as the request progresses, and also add logic in our reducers to prevent cases like loading data twice.

Let's update our `postsSlice` to use this pattern to track loading state for a "fetch posts" request. We'll switch our state from being an array of posts by itself, to look like `{posts, status, error}`. We'll also remove the old sample post entries from our initial state. As part of this change, we also need to change any uses of `state` as an array to be `state.posts` instead, because the array is now one level deeper:

features/posts/postsSlice.js

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

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

const postsSlice = createSlice({
  name: 'posts',
  initialState,
  reducers: {
    postAdded: {
      reducer(state, action) {
        state.posts.push(action.payload)
      },
      prepare(title, content, userId) {
```

```
        // omit prepare logic
      }
    },
    reactionAdded(state, action) {
      const { postId, reaction } = action.payload
      const existingPost = state.posts.find(post => post.id
=== postId)
      if (existingPost) {
        existingPost.reactions[reaction]++
      }
    },
    postUpdated(state, action) {
      const { id, title, content } = action.payload
      const existingPost = state.posts.find(post => post.id
=== id)
      if (existingPost) {
        existingPost.title = title
        existingPost.content = content
      }
    }
  }
})

export const { postAdded, postUpdated, reactionAdded } =
postsSlice.actions

export default postsSlice.reducer

export const selectAllPosts = state => state.posts.posts

export const selectPostById = (state, postId) =>
  state.posts.posts.find(post => post.id === postId)
```

Yes, this *does* mean that we now have a nested object path that looks like `state.posts.posts`, which is somewhat repetitive and silly :) We *could* change the nested array name to be `items` or `data` or something if we wanted to avoid that, but we'll leave it as-is for now.

Fetching Data with `createAsyncThunk`

Redux Toolkit's `createAsyncThunk` API generates thunks that automatically dispatch those "start/success/failure" actions for you.

Let's start by adding a thunk that will make an AJAX call to retrieve a list of posts. We'll import the `client` utility from the `src/api` folder, and use that to make a request to `'/fakeApi/posts'`.

features/posts/postsSlice

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

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

export const fetchPosts =
createAsyncThunk('posts/fetchPosts', async () => {
  const response = await client.get('/fakeApi/posts')
  return response.data
})
```

`createAsyncThunk` accepts two arguments:

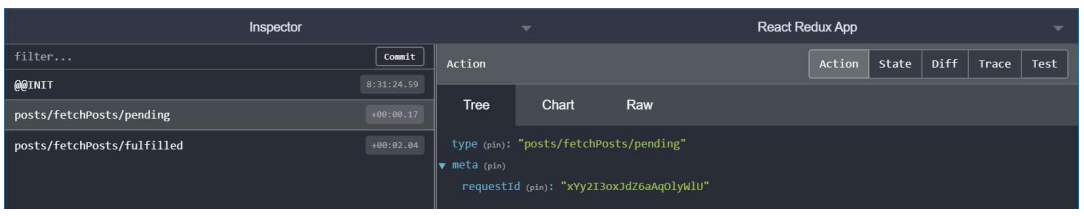
- A string that will be used as the prefix for the generated action types

- A "payload creator" callback function that should return a `Promise` containing some data, or a rejected `Promise` with an error

The payload creator will usually make an AJAX call of some kind, and can either return the `Promise` from the AJAX call directly, or extract some data from the API response and return that. We typically write this using the JS `async/await` syntax, which lets us write functions that use `Promise`s while using standard `try/catch` logic instead of `somePromise.then()` chains.

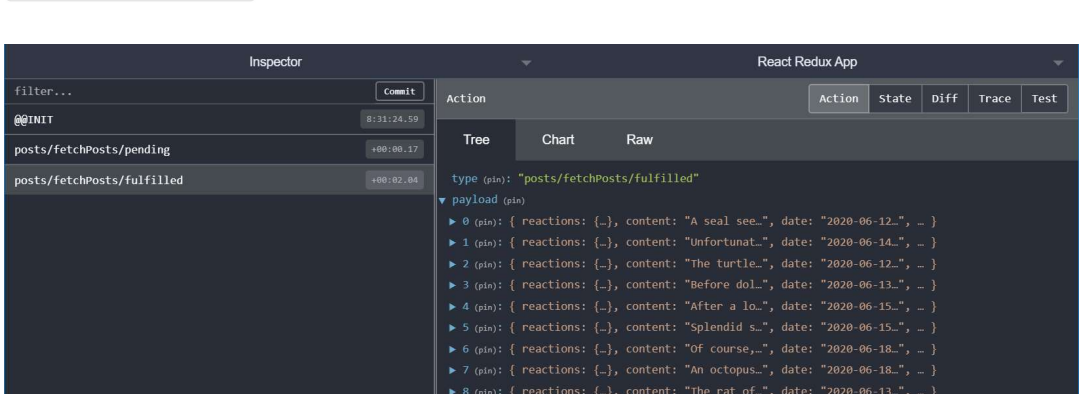
In this case, we pass in `'posts/fetchPosts'` as the action type prefix. Our payload creation callback waits for the API call to return a response. The response object looks like `{data: []}`, and we want our dispatched Redux action to have a payload that is *just* the array of posts. So, we extract `response.data`, and return that from the callback.

If we try calling `dispatch(fetchPosts())`, the `fetchPosts` thunk will first dispatch an action type of `'posts/fetchPosts/pending'`:



We can listen for this action in our reducer and mark the request status as `'loading'`.

Once the `Promise` resolves, the `fetchPosts` thunk takes the `response.data` array we returned from the callback, and dispatches a `'posts/fetchPosts/fulfilled'` action containing the posts array as `action.payload`:



Dispatching Thunks from Components

So, let's update our `<PostsList>` component to actually fetch this data automatically for us.

We'll import the `fetchPosts` thunk into the component. Like all of our other action creators, we have to dispatch it, so we'll also need to add the `useDispatch` hook. Since we want to fetch this data when `<PostsList>` mounts, we need to import the React `useEffect` hook:

features/posts/PostsList.js

```
import React, { useEffect } from 'react'
import { useSelector, useDispatch } from 'react-redux'
// omit other imports
import { selectAllPosts, fetchPosts } from './postsSlice'

export const PostsList = () => {
  const dispatch = useDispatch()
  const posts = useSelector(selectAllPosts)

  const postStatus = useSelector(state =>
state.posts.status)

  useEffect(() => {
```

```
    if (postStatus === 'idle') {
      dispatch(fetchPosts())
    }
  }, [postStatus, dispatch])

  // omit rendering logic
}
```

It's important that we only try to fetch the list of posts once. If we do it every time the `<PostsList>` component renders, or is re-created because we've switched between views, we might end up fetching the posts several times. We can use the `posts.status` enum to help decide if we need to actually start fetching, by selecting that into the component and only starting the fetch if the status is `'idle'`.

Reducers and Loading Actions

Next up, we need to handle both these actions in our reducers. This requires a bit deeper look at the `createSlice` API we've been using.

We've already seen that `createSlice` will generate an action creator for every reducer function we define in the `reducers` field, and that the generated action types include the name of the slice, like:

```
console.log(
  postUpdated({ id: '123', title: 'First Post', content:
'Some text here' })
)
/*
{
  type: 'posts/postUpdated',
  payload: {
    id: '123',
    title: 'First Post',
    content: 'Some text here'
  }
}
*/
```

However, there are times when a slice reducer needs to respond to *other* actions that weren't defined as part of this slice's `reducers` field. We can do that using the slice `extraReducers` field instead.

The `extraReducers` option should be a function that receives a parameter called `builder`. The `builder` object provides methods that let us define additional case reducers that will run in response to actions defined outside of the slice. We'll use `builder.addCase(actionCreator, reducer)` to handle each of the actions dispatched by our async thunks.

▶ **Detailed Explanation: Adding Extra Reducers to Slices**

In this case, we need to listen for the "pending" and "fulfilled" action types dispatched by our `fetchPosts` thunk. Those action creators are attached to our actual `fetchPost` function, and we can pass those to `extraReducers` to listen for those actions:

```
export const fetchPosts =
createAsyncThunk('posts/fetchPosts', async () => {
  const response = await client.get('/fakeApi/posts')
  return response.data
})

const postsSlice = createSlice({
  name: 'posts',
  initialState,
```

```
  reducers: {
    // omit existing reducers here
  },
  extraReducers(builder) {
    builder
      .addCase(fetchPosts.pending, (state, action) => {
        state.status = 'loading'
      })
      .addCase(fetchPosts.fulfilled, (state, action) => {
        state.status = 'succeeded'
        // Add any fetched posts to the array
        state.posts = state.posts.concat(action.payload)
      })
      .addCase(fetchPosts.rejected, (state, action) => {
        state.status = 'failed'
        state.error = action.error.message
      })
  }
})
```

We'll handle all three action types that could be dispatched by the thunk, based on the `Promise` we returned:

- When the request starts, we'll set the `status` enum to `'loading'`
- If the request succeeds, we mark the `status` as `'succeeded'`, and add the fetched posts to `state.posts`
- If the request fails, we'll mark the `status` as `'failed'`, and save any error message into the state so we can display it

Displaying Loading State

Our `<PostsList>` component is already checking for any updates to the posts that are stored in Redux, and rerendering itself any time that list changes. So, if we refresh the page, we should see a random set of posts from our fake API show up on screen:

The fake API we're using returns data immediately. However, a real API call will probably take some time to return a response. It's usually a good idea to show some kind of "loading..." indicator in the UI so the user knows we're waiting for data.

We can update our `<PostsList>` to show a different bit of UI based on the `state.posts.status` enum: a spinner if we're loading, an error message if it failed, or the actual posts list if we have the data. While we're at it, this is probably a good time to extract a `<PostExcerpt>` component to encapsulate the rendering for one item in the list as well.

The result might look like this:

features/posts/PostsList.js

```
import { Spinner } from '../../components/Spinner'
import { PostAuthor } from './PostAuthor'
import { TimeAgo } from './TimeAgo'
import { ReactionButtons } from './ReactionButtons'
import { selectAllPosts, fetchPosts } from './postsSlice'

const PostExcerpt = ({ post }) => {
  return (
    <article className="post-excerpt">
      <h3>{post.title}</h3>
      <div>
        <PostAuthor userId={post.user} />
        <TimeAgo timestamp={post.date} />
      </div>
      <p className="post-content">{post.content.substring(0,
100)}</p>
```

```
      <ReactionButtons post={post} />
      <Link to={`/posts/${post.id}`} className="button
muted-button">
        View Post
      </Link>
    </article>
  )
}

export const PostsList = () => {
  const dispatch = useDispatch()
  const posts = useSelector(selectAllPosts)

  const postStatus = useSelector(state =>
state.posts.status)
  const error = useSelector(state => state.posts.error)

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

  let content

  if (postStatus === 'loading') {
    content = <Spinner text="Loading..." />
  } else if (postStatus === 'succeeded') {
    // Sort posts in reverse chronological order by datetime
string
    const orderedPosts = posts
      .slice()
      .sort((a, b) => b.date.localeCompare(a.date))

    content = orderedPosts.map(post => (
      <PostExcerpt key={post.id} post={post} />
    ))
  } else if (postStatus === 'failed') {
    content = <div>{error}</div>
  }

  return (
    <section className="posts-list">
      <h2>Posts</h2>
      {content}
    </section>
  )
}
```

You might notice that the API calls are taking a while to complete, and that the loading spinner is staying on screen for a couple seconds. Our mock API server is configured to add a 2-second delay to all responses, specifically to help visualize times when there's a loading spinner visible. If you want to change this behavior, you can open up `api/server.js`, and alter this line:

api/server.js

```
// Add an extra delay to all endpoints, so loading spinners
show up.
const ARTIFICIAL_DELAY_MS = 2000
```

Feel free to turn that on and off as we go if you want the API calls to complete faster.

Loading Users

We're now fetching and displaying our list of posts. But, if we look at the posts, there's a problem: they all now say "Unknown author" as the authors:

This is because the post entries are being randomly generated by the fake API server, which also randomly generates a set of fake users every time we reload the page. We need to update our users slice to fetch those users when the application starts.

Like last time, we'll create another async thunk to get the users from the API and return them, then handle the `fulfilled` action in the `extraReducers` slice field. We'll skip worrying about loading state for now:

features/users/usersSlice.js

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

const initialState = []

export const fetchUsers =
createAsyncThunk('users/fetchUsers', async () => {
  const response = await client.get('/fakeApi/users')
  return response.data
})

const usersSlice = createSlice({
  name: 'users',
  initialState,
  reducers: {},
  extraReducers(builder) {
    builder.addCase(fetchUsers.fulfilled, (state, action) =>
{
      return action.payload
    })
  }
})

export default usersSlice.reducer
```

You may have noticed that this time the case reducer isn't using the `state` variable at all. Instead, we're returning the `action.payload` directly. Immer lets us update state in two ways: either *mutating* the existing state value, or *returning* a new result. If we return a new value, that will replace the existing state completely with whatever we return. (Note that if you want to manually return a new value, it's up to you to write any immutable update logic that might be needed.)

In this case, the initial state was an empty array, and we probably could have done `state.push(...action.payload)` to mutate it. But, in our case we really want to replace the list of users with whatever the server returned, and this avoids any chance of accidentally duplicating the list of users in state.

> **INFO**
>
> To learn more about how state updates with Immer work, see the "Writing Reducers with Immer" guide in the RTK docs.

We only need to fetch the list of users once, and we want to do it right when the application starts. We can do that in our `index.js` file, and

directly dispatch the `fetchUsers` thunk because we have the `store` right there:

index.js

```
// omit other imports

import store from './app/store'
import { fetchUsers } from './features/users/usersSlice'

import { worker } from './api/server'

async function main() {
  // Start our mock API server
  await worker.start({ onUnhandledRequest: 'bypass' })

  store.dispatch(fetchUsers())

  ReactDOM.render(
    <React.StrictMode>
      <Provider store={store}>
        <App />
      </Provider>
    </React.StrictMode>,
    document.getElementById('root')
  )
}
main()
```

Now, each of the posts should be showing a username again, and we should also have that same list of users shown in the "Author" dropdown in our `<AddPostForm>`.

Adding New Posts

We have one more step for this section. When we add a new post from the `<AddPostForm>`, that post is only getting added to the Redux store inside our app. We need to actually make an API call that will create the new post entry in our fake API server instead, so that it's "saved". (Since this is a fake API, the new post won't persist if we reload the page, but if we had a real backend server it would be available next time we reload.)

Sending Data with Thunks

We can use `createAsyncThunk` to help with sending data, not just fetching it. We'll create a thunk that accepts the values from our `<AddPostForm>` as an argument, and makes an HTTP POST call to the fake API to save the data.

In the process, we're going to change how we work with the new post object in our reducers. Currently, our `postsSlice` is creating a new post object in the `prepare` callback for `postAdded`, and generating a new unique ID for that post. In most apps that save data to a server, the server will take care of generating unique IDs and filling out any extra fields, and will usually return the completed data in its response. So, we can send a request body like `{ title, content, user: userId }` to the server, and then take the complete post object it sends back and add it to our `postsSlice` state.

features/posts/postsSlice.js

```
export const addNewPost = createAsyncThunk(
  'posts/addNewPost',
  // The payload creator receives the partial `{title,
content, user}` object
  async initialPost => {
```

```
    // We send the initial data to the fake API server
    const response = await client.post('/fakeApi/posts',
initialPost)
    // The response includes the complete post object,
including unique ID
    return response.data
  }
)

const postsSlice = createSlice({
  name: 'posts',
  initialState,
  reducers: {
    // The existing `postAdded` reducer and prepare callback
were deleted
    reactionAdded(state, action) {}, // omit logic
    postUpdated(state, action) {} // omit logic
  },
  extraReducers(builder) {
    // omit posts loading reducers
    builder.addCase(addNewPost.fulfilled, (state, action) =>
{
      // We can directly add the new post object to our
posts array
      state.posts.push(action.payload)
    })
  }
})
```

Checking Thunk Results in Components

Finally, we'll update `<AddPostForm>` to dispatch the `addNewPost` thunk instead of the old `postAdded` action. Since this is another API call to the server, it will take some time and *could* fail. The `addNewPost()` thunk will automatically dispatch its `pending/fulfilled/rejected` actions to the Redux store, which we're already handling. We *could* track the request status in `postsSlice` using a second loading enum if we wanted to, but for this example let's keep the loading state tracking limited to the component.

It would be good if we can at least disable the "Save Post" button while we're waiting for the request, so the user can't accidentally try to save a post twice. If the request fails, we might also want to show an error message here in the form, or perhaps just log it to the console.

We can have our component logic wait for the async thunk to finish, and check the result when it's done:

features/posts/AddPostForm.js

```
import React, { useState } from 'react'
import { useDispatch, useSelector } from 'react-redux'

import { addNewPost } from './postsSlice'

export const AddPostForm = () => {
  const [title, setTitle] = useState('')
  const [content, setContent] = useState('')
  const [userId, setUserId] = useState('')
  const [addRequestStatus, setAddRequestStatus] =
useState('idle')

  // omit useSelectors and change handlers

  const canSave =
    [title, content, userId].every(Boolean) &&
addRequestStatus === 'idle'

  const onSavePostClicked = async () => {
```

```
    if (canSave) {
      try {
        setAddRequestStatus('pending')
        await dispatch(addNewPost({ title, content, user:
userId })).unwrap()
        setTitle('')
        setContent('')
        setUserId('')
      } catch (err) {
        console.error('Failed to save the post: ', err)
      } finally {
        setAddRequestStatus('idle')
      }
    }
  }

  // omit rendering logic
}
```

We can add a loading status enum field as a React `useState` hook, similar to how we're tracking loading state in `postsSlice` for fetching posts. In this case, we just want to know if the request is in progress or not.

When we call `dispatch(addNewPost())`, the async thunk returns a `Promise` from `dispatch`. We can `await` that promise here to know when the thunk has finished its request. But, we don't yet know if that request succeeded or failed.

`createAsyncThunk` handles any errors internally, so that we don't see any messages about "rejected Promises" in our logs. It then returns the final action it dispatched: either the `fulfilled` action if it succeeded, or the `rejected` action if it failed.

However, it's common to want to write logic that looks at the success or failure of the actual request that was made. Redux Toolkit adds a `.unwrap()` function to the returned `Promise`, which will return a new `Promise` that either has the actual `action.payload` value from a `fulfilled` action, or throws an error if it's the `rejected` action. This lets us handle success and failure in the component using normal `try/catch` logic. So, we'll clear out the input fields to reset the form if the post was successfully created, and log the error to the console if it failed.

If you want to see what happens when the `addNewPost` API call fails, try creating a new post where the "Content" field only has the word "error" (without quotes). The server will see that and send back a failed response, so you should see a message logged to the console.

What You've Learned

Async logic and data fetching are always a complex topic. As you've seen, Redux Toolkit includes some tools to automate the typical Redux data fetching patterns.

Here's what our app looks like now that we're fetching data from that fake API:

App.js

```
import React from 'react'
import {
  BrowserRouter as Router,
  Switch,
  Route,
  Redirect,
} from 'react-router-dom'

import { Navbar } from './app/Navbar'

import { PostsList } from './features/posts/Pos
import { AddPostForm } from './features/posts/A
import { EditPostForm } from './features/posts/
import { SinglePostPage } from './features/post

function App() {
  return (
    <Router>
      <Navbar />
      <div className="App">
```

Click to Run

Open S

As a reminder, here's what we covered in this section:

SUMMARY

- **You can write reusable "selector" functions to encapsulate reading values from the Redux state**
  - Selectors are functions that get the Redux `state` as an argument, and return some data
- **Redux uses plugins called "middleware" to enable async logic**
  - The standard async middleware is called `redux-thunk`, which is included in Redux Toolkit
  - Thunk functions receive `dispatch` and `getState` as arguments, and can use those as part of async logic
- **You can dispatch additional actions to help track the loading status of an API call**
  - The typical pattern is dispatching a "pending" action before the call, then either a "success" containing the data or a "failure" action containing the error
  - Loading state should usually be stored as an enum, like `'idle' | 'loading' | 'succeeded' | 'failed'`
- **Redux Toolkit has a `createAsyncThunk` API that dispatches these actions for you**
  - `createAsyncThunk` accepts a "payload creator" callback that should return a `Promise`, and generates `pending/fulfilled/rejected` action types automatically
  - Generated action creators like `fetchPosts` dispatch those actions based on the `Promise` you return
  - You can listen for these action types in `createSlice` using the `extraReducers` field, and update the state in reducers based on those actions.
  - Action creators can be used to automatically fill in the keys of the `extraReducers` object so the slice knows what actions to listen for.
  - Thunks can return promises. For `createAsyncThunk` specifically, you can `await dispatch(someThunk()).unwrap()` to handle the request success or failure at the component level.

What's Next?

We've got one more set of topics to cover the core Redux Toolkit APIs and usage patterns. In Part 6: Performance and Normalizing Data, we'll look at how Redux usage affects React performance, and some ways we can optimize our application for improved performance.

Last updated on ***Nov 15, 2023***