Redux

Redux News Reader

This project—a news reader in which users can view and comment on various articles—gives you an opportunity to practice using Redux Toolkit’s createAsyncThunk and createSlice utilities to add asynchronous functionality to your Redux applications.

Currently, the app fetches and displays a list of all articles in their preview form. It also fetches and displays the current article, which can be selected and changed by the user. The code for these features is accessible in the **features/articlePreviews** and **features/currentArticle** directories respectively.

Your task will be to complete the comments feature. Whenever the featured article changes, you will asynchronously fetch the comments for that article and add them to your store so they display under the article. Likewise, when a user submits a new comment, you will submit it to the server via an asynchronous request and display it in the article’s list of comments once it has been successfully created.

Before you get started, spend some time familiarizing yourself with the project’s starting code. In particular, take note of the way the project employs createSlice and createAsyncThunk in **currentArticleSlice.js** and **articlePreviewsSlice.js**, as your work on the comments slice will resemble these files.

This lesson uses [Mock Service Worker](https://mswjs.io/) (MSW) to replicate the functionality of an external API. To use MSW, you’ll want to use Google Chrome and [enable third-party cookies](https://support.google.com/chrome/answer/95647?co=GENIE.Platform%3DDesktop&hl=en).

**Tasks**

15/16 Complete

Mark the tasks as complete by checking them off

**Write Your First Asynchronous Action Creator**

1.

We’re going to start by using createAsyncThunk to create an asynchronous action creator, loadCommentsForArticleId, that fetches all the comments for a particular article.

In **commentsSlice.js**, import createAsyncThunk from Redux Toolkit.

Import createAsyncThunk from Redux Toolkit using the following syntax. Leave the import of createSlice in place, as we use that function to generate a slice for the comments.

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

2.

Now that you’ve imported createAsyncThunk, let’s use it to define an asynchronous action creator that fetches all the comments for a particular article.

* Call createAsyncThunk with two arguments—an action type string, and an asynchronous callback that takes an article id as its first argument.
* Store the result in a constant called loadCommentsForArticleId and export it.

Your call to createAsyncThunk should look something like this:

export const asyncActionCreator = createAsyncThunk(  
  'nameOfSlice/nameOfAction',  
  async (arg) => {  
    // do some asynchronous operations here   
  });

Remember that the your action type string should take the form nameOfSlice/nameOfAction. In this case, the slice is 'comments' and the action is 'loadCommentsForArticleId'.

3.

Now let’s implement the body of the asynchronous callback passed to createAsyncThunk(). This callback should make an asynchronous request to our news API for the comments associated with the given article id.

The route for fetching comments for an article by id is 'api/articles/ID/comments'. For example, to fetch the comments for an article with an id = 123 using the [Fetch API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API), you could write

fetch('api/articles/123/comments')

Inside the asynchronous callback passed to createAsyncThunk():

1. Call fetch() with the proper route for fetching the comments of the given article id parameter.
2. Since fetch() is asynchronous, you’ll want to await the result and store it in a variable called response.
3. fetch returns a Promise containing the HTTP response to the request. To get the actual JSON data from the response body, call .json() on response. .json() is also asynchronous, so you’ll once again want to await the result and store it in a variable called json.
4. Finally, return json.

Your thunk should look something like this:

const response = await fetch(`api/articles/${id}/comments`);  
const json = await response.json();  
return json;

Make sure your thunk returns json!

**Add Extra Reducers to the Comments Slice**

4.

Now that you’ve written loadCommentsForArticleId, you should update commentsSlice to store comments. There are several ways you could choose to organize your slice, but we recommend storing comments in an object keyed by article IDs, where each article id key corresponds to an array of comments belonging to that article. For example:

{  
  123: ['Great article!' , 'I disagree.']  
  456: ['This is some great writing.'],  
  ...   
}

Since comments always belong to articles and can only belong to a single article, this approach will simplify the process of retrieving all the comments for a particular article.

In **commentsSlice.js**, add a byArticleId property to initialState, and set it equal to an empty object.

Our desired slice structure looks like:

byArticleId: {  
    articleId1: [  
      {comment},  
      {comment},  
    ],  
    articleId2: [  
      {comment},  
    ]   
  }

5.

Your asynchronous action creator, loadCommentsForArticleId, will dispatch an action for each of the pending/fulfilled/rejected promise lifecycle states, and your slice should keep track of which state the asynchronous request is in so that it can display loading and error states as appropriate.

To make your slice track these promise lifecycle states, you should first update your slice’s initial state to include two booleans: isLoadingComments and failedToLoadComments, and initialize them appropriately.

Since the app is neither loading comments nor failing to load comments at initialization, both properties should be set to false.

6.

By default, slices produced by calls to createSlice only respond to actions that were also created by createSlice. This means that the loadCommentsForArticleId thunk you wrote won’t have any effect on the store until you make the commentsSlice respond to it by using the extraReducers property.

In **commentsSlice.js**

1. Add anextraReducers property to the configurations options object passed to createSlice. You can use the [map object notation](https://redux-toolkit.js.org/api/createSlice#the-extrareducers-map-object-notation) you learned in the lesson, or study the documentation for [builder callback notation](https://redux-toolkit.js.org/api/createSlice#the-extrareducers-builder-callback-notation) and use that instead.
2. For now, set the property to an empty object.

At this point, your configurations options object should have name, initialState and extraReducers properties.

7.

Now that you’ve added the extraReducers property to your slice configurations, you’ll want to add a reducer for each of the three actions dispatched by loadCommentsForArticleId:

1. The pending promise lifecycle action
2. The rejected promise lifecycle action
3. The fulfilled promise lifecycle action. Note that in this case, action.payload is a comment object with an articleId property you can use to add the comment to the correct article’s comment list in state.

Each one should update the state accordingly.

You can access each of the pending/fulfilled/rejected actions like so:

* yourAsynchronousActionCreator.pending
* yourAsynchronousActionCreator.fulfilled
* yourAsynchronousActionCreator.rejected

The fulfilled reducer should add the newly fetched comments to the byArticleId object you added to initialState, and set isLoadingComments and failedToLoadComments to false.

The pending and rejected actions should just set those booleans as appropriate.

**Display Comments for the Current Article**

8.

Now that you’ve written your comments slice, you’ll want to actually display the comments for the current article. Navigate to **Comments.js**, the component that will handle all the comment-related logic. Note that we’ve already imported and used the selectCurrentArticle selector from the **currentArticleSlice** to define the current article in the constant article.

We’ve also defined two constants, comments and commentsAreLoading, and initialized them with temporary values that you will overwrite using the selectors imported from the comments slice.

* Replace the empty array currently assigned to comments with the comments currently in state
* Replace the false boolean currently assigned to commentsAreLoading with the state value representing whether or not there is currently a pending request to fetch comments.

For a refresher on how useSelector works, refer to the React Redux [hook documentation](https://react-redux.js.org/api/hooks#useselector).

9.

In order to make sure the store includes all the comments associated with the current article, we should fetch all the comments for the current article any time the the current article changes. Once we have the comments for the current article, we’ll pass them to CommentList to make them show up on the page.

In **Comments.js**:

1. Using useEffect, dispatch loadCommentsForArticleId any time article changes and only if article is not undefined.
2. Define a constant, commentsForArticleId, which should be an empty array when article is undefined and otherwise should be equal to comments[article.id].
3. Replace the empty array we’ve passed as the CommentList‘s comments prop with commentsForArticleId.

For a refresher on how useEffect works, refer to the hooks [documentation](https://reactjs.org/docs/hooks-reference.html#useeffect). The function you pass to useEffect should call loadCommentsForArticleId with article.id as an argument, and you’ll want to include dispatch and article in the dependency array to ensure you fetch comments any time article changes.

10.

Now that you’ve taken care of passing the comments to the CommentList component, you need to make that component render the comments it receives.

Inside the <ul> of the CommentList component’s return statement

* Map over the comments prop and render a Comment (we’ve imported this component for you) for each value.
* Each Comment component needs to be passed a comment prop.

If you refresh your browser now and click on an article, you should see that article’s comments displayed below it!

Your logic should look something like this:

if (!comments) {  
  return null;  
}   
   
return (  
  <ul>  
    {  
      comments.map(comment => {  
        // Return a Comment component  
      }  
    }  
  </ul>  
)

**Write postCommentForArticleId**

11.

Now that you’ve implemented one asynchronous feature, you’ll go through a similar process to add another—a form for creating new comments for the current article.

First, back in **commentsSlice.js**, you’ll write another asynchronous action creator, postCommentForArticleId, which will be called like so:

postCommentForArticleId({  
  articleId: 1,  
  comment: "This article is great!"  
}}

Inside **commentsSlice.js**

* Declare a new exported variable called postCommentForArticleId.
* Call createAsyncThunk with two arguments—an action type string, and an asynchronous callback—and store the result in postCommentForArticleId.
* The asynchronous callback should accept an object with two properties–articleId and comment–as a parameter. You should destructure the object in your function definition so that you can refer to articleId and comment when you implement the function body.

Your call to createAsyncThunk should look something like this:

export const asyncActionCreator = createAsyncThunk(  
  'nameOfSlice/nameOfAction',  
  async ({ prop1, prop2 }) => {  
    // do some asynchronous operations here   
  }  
);

In this case, the name of the slice is 'comments', and the name of the action is 'postCommentForArticleId'.

12.

Now let’s implement the body of the asynchronous callback to make an asynchronous POST request to the news API.

To make a POST request, we can again use the fetch() method but we must specify that we want to make a POST request. We must also pass along the comment value included in the parameter object (formatted as a stringified object).

Inside the asynchronous callback passed to createAsyncThunk(),

1. Declare a new variable called requestBody. Then, call JSON.stringify() and pass in an object with a single property, comment, corresponding to the text of the new comment included in the parameter object. Assign the result to requestBody.
2. Call the fetch() method to make a request to 'api/articles/ID/comments', replacing 'ID' with the articleId value included in the parameter object.
3. Pass an options object as the second argument to fetch() after the URL. This object should have a method key with the value equal to the string 'POST'.
4. Add a body key to the options object with the value equal requestBody.
5. Since fetch is asynchronous, you’ll want to await the result and store it in a variable called response.
6. To get the actual JSON data from the response body, call .json() on response. .json() is also asynchronous, so you’ll once again want to await the result and store it in a variable called json.
7. Finally, return json.

Your thunk should do something like this:

const response = await fetch(PATH\_TO\_RESOURCE, {  
  method: 'POST',  
  body: REQUEST\_BODY\_HERE  
});  
const json = await response.json();  
return json

Make sure your thunk returns json!

13.

Like all action creators generated by createAsyncThunk, postCommentForArticleId will dispatch actions for each of the pending/fulfilled/rejected promise lifecycle states. In order to make our app reflect these changing states, we have to keep track of them in the store.

In order to do that, add two booleans—createCommentIsPending and failedToCreateComment—to initialState, and set their initial values appropriately.

Since the app is not attempting to create a comment at initialization, both properties should be set to false.

14.

Now modify the extraReducers property of your slice configuration options by adding a reducer for each of the three promise lifecyle actions dispatched by postCommentsForArticleId:

1. The pending promise lifecycle action
2. The rejected promise lifecycle action
3. The fulfilled promise lifecycle action. Note that in this case, action.payload will be a comment object including an articleId that you can use to add the comment object to correct article’s comment list in state.

Each one should update the state accordingly.

You can access each of the pending/fulfilled/rejected actions like so:

* yourAsynchronousActionCreator.pending
* yourAsynchronousActionCreator.fulfilled
* yourAsynchronousActionCreator.rejected
* The fulfilled reducer should add the newly created comment to the byArticleId object you added to initialState, and set createCommentIsPending and failedToCreateComment to false.
* The pending and rejected actions should just set those booleans as appropriate.

**Improve Comment Form UX**

15.

Well done! You’ve created two asynchronous actions—one for getting comments and one for adding a posting new ones—and handled them using extraReducers within createSlice‘s configuration object. Just two more steps to go.

First, you’re going to connect the CommentForm we’ve provided to your new asynchronous action creator

In **CommentForm.js**:

1. Import postCommentForArticleId from the comments slice.
2. In handleSubmit, dispatch postCommentForArticleId passing in an object with articleId and comment.

At this point, you should be able to create new comments with the comment form. Try it out to test your work!

To import postCommentForArticleId, add it to the import statement that includes createCommentIsPending.

To dispatch your asynchronous action, you will write code that looks like this:

dispatch(  
  asynchronousActionCreator({  
    /\* action creator arguments here\*/  
  })  
)

16.

Good job! Your last task is to improve the user experience slightly by disabling the submit button when a request to create a new comment is pending. This will prevent users from accidentally creating the same comment twice.

In **CommentForm.js**:

1. Define a constant, isCreatePending, using useSelector and the imported selector createCommentIsPending.
2. Add a [disabled](https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes/disabled) attribute to the form’s button and set it equal to isCreatePending. Refresh the browser and try to create the same comment twice by double clicking. You’ll see that it’s now impossible to do so.

To add a disabled attribute to a button, you’ll want to add disabled={BOOLEAN EXPRESSION} to the button’s opening tag.