GLAB 320.11.1 - Redux State Management

Version 1.0, 04/04/23
React 18.2.0, Redux Toolkit 1.9.3

Introduction

In this guided lab, we will explore the Redux Toolkit (RTK) through building an example application that allows us to create, view, edit, and react to content posts in a "social media app" scenario. We will optimize our application's rendering process, create asynchronous thunk functions to handle API requests, and normalize our application state.

This lab is structured in parts; each part is designed so that it can be completed independently and in separate sittings. It is recommended to split this lab over the course of multiple days in order to allow the content to digest.

- Part 0: Explore the Initial Project
- Part 1: Redux Data Flow
- Part 2: Using Redux Data
- Part 3: Data Fetching
- Part 4: Performance Optimization

CodeSandbox links have been provided at the end of each part, and can be used to skip ahead to parts most relevant to a particular group of learners.

This activity is adapted from the Redux Essentials tutorial series.

Objectives

- Integrate Redux into a React application.
- Implement the basic Redux data flow pattern.
- Use the Redux Toolkit APIs to simplify code and reduce boilerplate.
- Implement basic Create, Read, Update, and Delete (CRUD) functionality within a React-Redux application using Redux Toolkit.
- Access an external API through the use of asynchronous thunk functions.
- Optimize application performance through the use of memoized selectors (which will be explained during the course of this lab activity).
- Optimize application performance through the use of normalized state.

Tools and Software

- A CodeSandbox account and a GitHub account
- React DevTools and Redux DevTools Chrome extensions

Instructions

To begin, open and fork this CodeSandbox.

You can also clone the project from this GitHub repository. After cloning the repo, you can install the tools necessary for the project with `npm install` and start it with `npm start`.

This pre-configured project already has React and Redux set up, includes some default styling, and has a fake REST API that will allow us to write API requests in our application. You will use this pre-configured project as the basis for writing your application's code.

You can continue the remainder of this lab activity within CodeSandbox, or locally within your code editor of choice.

Part 0: Explore the Initial Project

Take a look at what the initial project contains, and familiarize yourself with both its structure and its contents. This app will eventually contain multiple interfaces for posting textual content and interacting with those posts.

If you load the app now, you should see a header and a welcome message. You can also open the Redux DevTools extension to see that the initial Redux state is empty.

Here is a breakdown of the current file tree:

- **`/public`** – Contains the HTML host page template and other static files like icons.
- **`/src`**
  - **`index.js`** – The entry point for the application. It renders the React-Redux `<Provider>` component and the main `<App>` component.
  - **`App.js`** – The main application component. Renders the top navbar and handles client-side routing for the other content.
  - **`index.css`** – Contains styles for the complete application.
  - **`/api`**
    - **`client.js`** – A small AJAX request client that allows us to make GET and POST requests.
    - **`server.js`** – Provides a fake REST API for our data. Our app will fetch data from these fake endpoints later.
  - **`/app`**

- `Navbar.js` – Renders the top header and nav content.
- `store.js` – Creates the Redux store instance.

Once you've finished this activity, you'll likely want to introduce Redux to your own projects. One way to create a new React + Redux project is by using the Redux templates for Create-React-App. These templates come with Redux Toolkit and React-Redux already configured, which allows you to get started writing your application's code without having to add Redux packages and set up the Redux store. If you are using a React framework like Next.js or Vite, there are more recommended ways to create a React + Redux project.

Part 1: Redux Data Flow

The main feature of this application will be a list of posts. As we continue, we will add several additional pieces to this feature, but our first goal is to show the list of post entries on screen.

Creating the Posts Slice

First, we will create a Redux "slice" that will contain the data for our posts. Once we have that data in the Redux store, we can create the React components to show the data on the page.

- Inside of `src`, create a new `features` folder.
- Put a `posts` folder inside of `features`.
- Add a new file named `postsSlice.js` inside of the `posts` folder.

Inside of this new file, we will use the RTK createSlice function to make a reducer function that will handle our posts data. Reducer functions need to have some initial data included so that the Redux store has those values loaded when the app starts.

For now, we'll create an array with some fake post objects inside so that we can begin adding UI elements.

Import `createSlice`, define an initial posts array, pass that to `createSlice`, and export the posts reducer function that `createSlice` has generated, as follows:

`features/posts/postsSlice.js`

JavaScript

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

const initialState = [
  { id: '1', title: 'First Post!', content: 'Hello!' },
```

```
  { id: '2', title: 'Second Post', content: 'More text' }
]

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

export default postsSlice.reducer
```

Every time we create a new slice in Redux, we need to add its reducer function to the Redux store. While our store is already being created, it does not yet have any data inside of it.

Open `app/store.js` and import the `postsReducer` function that we just created. Update the call to `configureStore` so that the `postsReducer` is passed as a reducer field named `posts`:

app/store.js

JavaScript

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

import postsReducer from '../features/posts/postsSlice'

export default configureStore({
  reducer: {
    posts: postsReducer
  }
})
```

This tells Redux that we want our top-level state object to have a field named posts inside, and all the data for `state.posts` will be updated by the `postsReducer` function when actions are dispatched.

Confirm this works by opening the Redux DevTools Extension and looking at the current state:

Now that we have some data in our store, we can create a React component that shows the list of posts.

Showing the Posts List

All of the code related to our posts feature should go in the `posts` folder, so create a new file within that folder called `PostsList.js`.

In order to render a list of posts, we'll need data from the Redux store. React components can read data from the store using the `useSelector` hook from the React-Redux library. The selector functions that you write will be called with the entire Redux `state` object as a parameter, and should return specific data that this component needs from the store.

The initial `PostsList` component will read the `state.posts` value from the Redux store, then loop over the array of posts and show each of them on screen:

`features/posts/PostsList.js`

JavaScript

```
import React from 'react'
import { useSelector } from 'react-redux'

export const PostsList = () => {
  const posts = useSelector(state => state.posts)

  const renderedPosts = posts.map(post => (
    <article className="post-excerpt" key={post.id}>
      <h3>{post.title}</h3>
      <p className="post-content">
        {post.content.substring(0, 100)}
      </p>
```

```
    </article>
  ))

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

Note that this component only shows the first 100 characters of a post's content (an excerpt). Now that we have a `PostsList` component, we need to render it within our application.

Import the `PostsList` component into `App.js`, and replace the existing welcome text with `<PostsList />`. Take the time to also wrap in a [React Fragment](), as we will be adding something else to the main page soon:

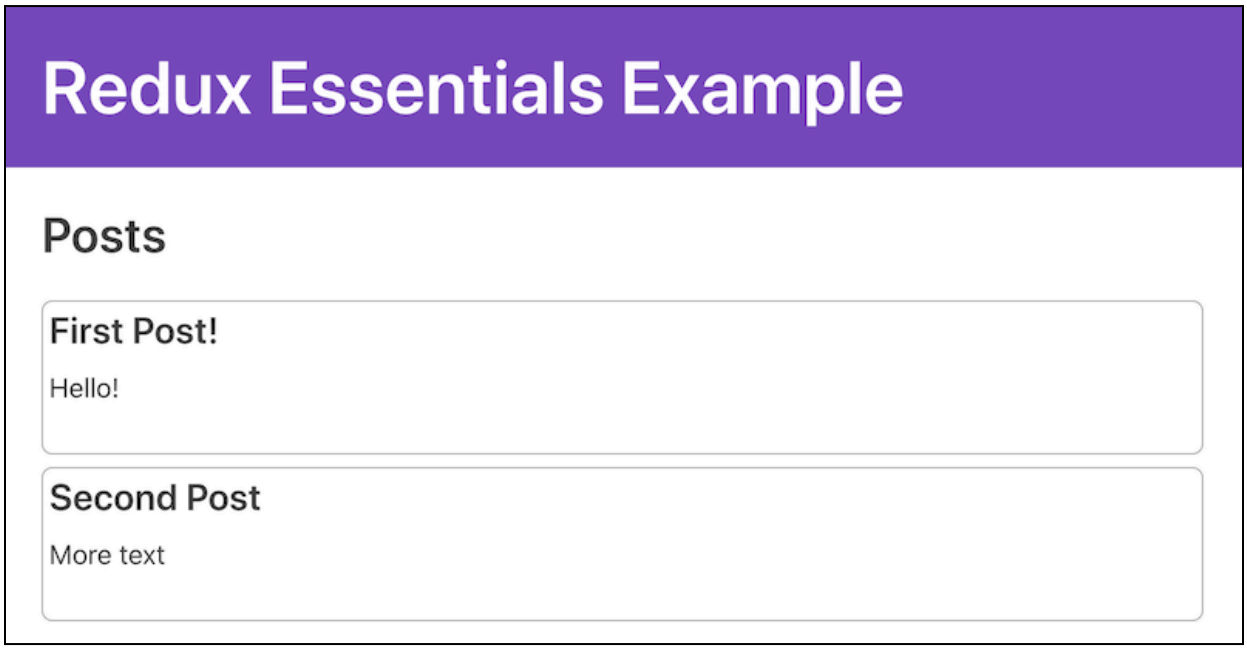
`App.js`

JavaScript

```
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/PostsList'

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

```
          <Route
            exact
            path="/"
            render={() => (
              <>
                <PostsList />
              </>
            )}
          />
          <Redirect to="/" />
        </Switch>
      </div>
    </Router>
  )
}
export default App
```

With our PostsList now included within the application, we should have content! Check the main page of your app, which should look like this:



We've successfully added some data to the Redux store, and shown it on screen within a React component. Next, we'll add a form that allows us to create and store new posts.

Adding New Posts

The posts that are currently visible within the application are hard-coded into our initial state. In order to create a more dynamic experience, we need to allow for the addition of new posts by our users.

We will create an “Add Post” form that allows users to write posts and save them. Being the creatively descriptive programmers we are, we’ll name the file for this component `AddPostForm.js` and put it in our `posts` folder.

First, we will create the empty form and add it to the page. We’ll add a text input for the post title, and a text area for the body of the post.

Note that, in the following code block, we use the `useState` hook from React. It is very important to recognize when to use and when not to use the Redux store to handle state information. In this case, the state that we’ll be tracking – title and content – is only relevant to the form component that we’re developing, not the global application. We will later send this information to our application through the use of a Redux action.

`features/posts/AddPostForm.js`

JavaScript

```
import React, { useState } from 'react'

export const AddPostForm = () => {
  const [title, setTitle] = useState('')
  const [content, setContent] = useState('')

  const onTitleChanged = e => setTitle(e.target.value)
  const onContentChanged = e => setContent(e.target.value)

  return (
    <section>
      <h2>Add a New Post</h2>
      <form>
        <label htmlFor="postTitle">Post Title:</label>
        <input
          type="text"
          id="postTitle"
          name="postTitle"
          value={title}
```

```
          onChange={onTitleChanged}
        />
        <label htmlFor="postContent">Content:</label>
        <textarea
          id="postContent"
          name="postContent"
          value={content}
          onChange={onContentChanged}
        />
        <button type="button">Save Post</button>
      </form>
    </section>
  )
}
```

Import this new component into `App.js` and add it above the `<PostsList />` component.

`App.js`

JavaScript

```
<Route
  exact
  path="/"
  render={() => (
    <>
      <AddPostForm />
      <PostsList />
    </>
  )}
/>
```

You should now see the form show up in the page right below the header, but interacting with it will not yet update our posts.

Saving Post Entries

Now, we need to update our Redux store with new posts entries.

To accomplish this, we need to update the posts slice.The posts slice is responsible for handling all updates to the posts data. Inside of the `createSlice` call, there's an object called `reducers`. Right now, that object is empty. We need to add a reducer function inside of there to handle the case of a post being added.

Inside of `reducers`, add a function named `postAdded`, which will receive two arguments: the current `state` value, and the `action` object that was dispatched. Since the posts slice only knows about the data it's responsible for, the `state` argument will be the array of posts by itself, and not the entire Redux state object.

The `action` object will have our new post entry as the `action.payload` field, and we'll put that new post object into the state array.

When we write the `postAdded` reducer function, `createSlice` will automatically generate an "action creator" function with the same name. We can export that action creator and use it in our UI components to dispatch the action when the user clicks "Save Post".

`features/posts/postsSlice.js`

JavaScript

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

export const { postAdded } = postsSlice.actions

export default postsSlice.reducer
```

DANGER – STATE MUTATIONS

Remember that **reducer functions must always create new state values immutably!**

It's safe to call mutating functions like `Array.push()` or modify object fields like `state.field = value` inside of `createSlice()`, because it uses the Immer library to convert those mutations into

safe immutable updates internally.

Do not try to mutate any state data outside of `createSlice`!

Dispatching the “Post Added” Action

Our `AddPostForm` has text inputs and a "Save Post" button, but the button doesn't do anything yet. We need to add a click handler that will dispatch the `postAdded` action creator and pass in a new post object containing the title and content the user wrote.

Our post objects also need to have an `id` field. Right now, our initial test posts are using some fake numbers for their IDs. We could write some code that would figure out what the next incrementing ID number should be, but it would be better if we generated a random unique ID instead. Redux Toolkit has a `nanoid` function we can use for that.

We'll talk more about generating IDs and dispatching actions in the next part of this process.

In order to dispatch actions from a component, we need access to the store's `dispatch` function. We get this by calling the `useDispatch` hook from React-Redux. We also need to import the `postAdded` action creator into this file.

Once we have the `dispatch` function available in our component, we can call `dispatch(postAdded())` in a click handler. We can take the title and content values from our React component `useState` hooks, generate a new ID, and put them together into a new post object that we pass to `postAdded()`.

`features/posts/AddPostForm.js`

JavaScript

```
import React, { useState } from 'react'
import { useDispatch } from 'react-redux'
import { nanoid } from '@reduxjs/toolkit'

import { postAdded } from './postsSlice'

export const AddPostForm = () => {
  const [title, setTitle] = useState('')
  const [content, setContent] = useState('')

  const dispatch = useDispatch()
```

```
  const onTitleChanged = e => setTitle(e.target.value)
  const onContentChanged = e => setContent(e.target.value)

  const onSavePostClicked = () => {
    if (title && content) {
      dispatch(
        postAdded({
          id: nanoid(),
          title,
          content
        })
      )
      setTitle('')
      setContent('')
    }
  }

  return (
    <section>
      <h2>Add a New Post</h2>
      <form>
        {/* omit form inputs */}
        <button type="button" onClick={onSavePostClicked}>
          Save Post
        </button>
      </form>
    </section>
  )
}
```

Now, try building a new post within the application by typing in a title and some content text. Click “Save Post,” and you should see a new item for that post show up in the posts list.

Congratulations! You’ve just built a working React + Redux app!

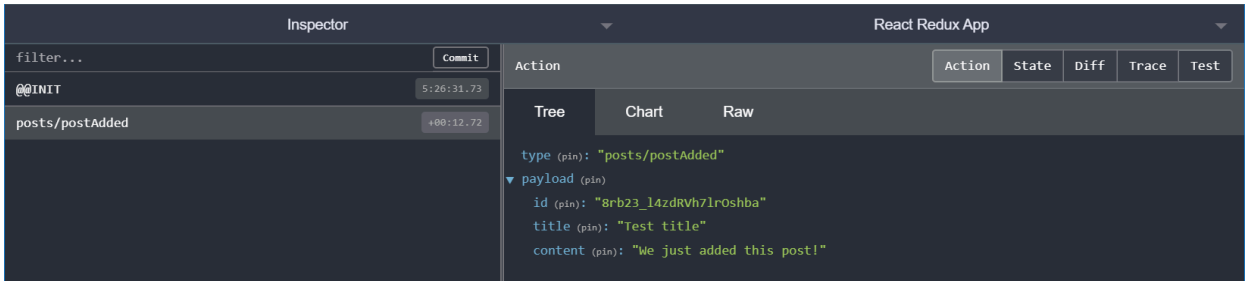
The application up to this point demonstrates the complete Redux data flow cycle:

- Our posts list read the initial set of posts from the store with `useSelector` and rendered the initial UI.
- We dispatched the `postAdded` action containing the data for the new post entry.

- The posts reducer saw the `postAdded` action, and updated the posts array with the new entry.
- The Redux store told the UI that some data had changed.
- The posts list read the updated posts array, and re-rendered itself to show the new post.

All the new features we'll add after this will follow the same basic patterns you've seen here: adding slices of state, writing reducer functions, dispatching actions, and rendering the UI based on data from the Redux store. Understanding this pattern is the key to mastering Redux.

We can check the Redux DevTools Extension to see the action we dispatched, and look at how the Redux state was updated in response to that action. If we click the "`posts/postAdded`" entry in the actions list, the "Action" tab should look like this:



The "Diff" tab should also show us that `state.posts` had one item added, which is at index 2.

Remember that our `AddPostForm` component has some React `useState` hooks inside, to keep track of the title and content values the user is typing in. As a reminder, **the Redux store should only contain data that's considered "global" for the application!** In this case, only the `AddPostForm` will need to know about the latest values for the input fields, so we want to keep that data in React component state instead of trying to keep the temporary data in the Redux store. When the user is done with the form, we dispatch a Redux action to update the store with the final values based on the user input.

Part One Recap

So far, we've reinforced the following:

- **Redux state is updated by "reducer functions."**
  - Reducers always calculate a new state immutably, by copying existing state values and modifying the copies with the new data.
  - The Redux Toolkit `createSlice` function generates "slice reducer" functions for you, and lets you write "mutating" code that is turned into safe immutable updates.

    - Those slice reducer functions are added to the `reducer` field in `configureStore`, and that defines the data and state field names inside the Redux store.
- **React components read data from the store with the `useSelector` hook.**
    - Selector functions receive the whole `state` object, and should return a value.
    - Selectors will re-run whenever the Redux store is updated, and if the data they return has changed, the component will re-render.
- **React components dispatch actions to update the store using the `useDispatch` hook.**
    - `createSlice` will generate action creator functions for each reducer we add to a slice.
    - Call `dispatch(someActionCreator())` in a component to dispatch an action.
    - Reducers will run, check to see if this action is relevant, and return a new state if appropriate.
    - Temporary data like form input values should be kept as React component state. Dispatch a Redux action to update the store when the user is done with the form.

Here's what your application should look like so far: Part One Completed CodeSandbox.

Part Two Preview

In the next section, we'll add some additional functionality to the application and handle some examples of how to work with data that is already inside of the Redux store.

Part 2: Using Redux Data

Now that you have gone through and familiarized yourself with the core steps to write Redux logic, we're going to use those same steps to add some new features to our social media feed that will make it more useful.

We will add the ability to view a single post, edit existing posts, show post author details and post timestamps, and use reaction buttons to add reactions to posts.

Showing Single Posts

Since we have the ability to add new posts to the Redux store, we can add some more features that use the post data in different ways.

Currently, our post entries are being shown in the main feed page, but if the text is too long, we only show an excerpt of the content. It would be helpful to have the ability to view a single post entry on its own page.

First, we need to add a new `SinglePostPage` component to our `posts` feature folder. We'll use React Router to show this component when the page URL looks like `/posts/123`, where the `123` part should be the ID of the post we want to show.

React Router will pass in a `match` object as a prop that contains the URL information we're looking for. When we set up the route to render this component, we're going to tell it to parse the second part of the URL as a variable named `postId`, and we can read that value from `match.params`.

Once we have that `postId` value, we can use it inside a selector function to find the right post object from the Redux store. We know that `state.posts` should be an array of all post objects, so we can use the `Array.find()` function to loop through the array and return the post entry with the ID we're looking for.

It's important to note that the component will re-render any time the value returned from `useSelector` changes to a new reference. Components should always try to select the smallest possible amount of data they need from the store, which will help ensure that it only renders when it actually needs to.

It's possible that we might not have a matching post entry in the store - maybe the user tried to type in the URL directly, or we don't have the right data loaded. If that happens, the `find()` function will return `undefined` instead of an actual post object. Our component needs to check for that and handle it by showing a "Post not found!" message on the page.

Assuming we do have the right post object in the store, `useSelector` will return that, and we can use it to render the title and content of the post in the page.

`features/posts/SinglePostPage.js`

JavaScript

```
import React from 'react'
import { useSelector } from 'react-redux'

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

  const post = useSelector(state =>
    state.posts.find(post => post.id === postId)
  )

  if (!post) {
```

```
    return (
      <section>
        <h2>Post not found!</h2>
      </section>
    )
  }

  return (
    <section>
      <article className="post">
        <h2>{post.title}</h2>
        <p className="post-content">{post.content}</p>
      </article>
    </section>
  )
}
```

You might notice that this looks fairly similar to the logic we have in the body of our `<PostsList>` component, where we loop over the whole `posts` array to show post excerpts on the main feed. We could try to extract a `Post` component that could be used in both places, but there are already some differences in how we're showing a post excerpt and the whole post. It's usually better to keep writing things separately for a while even if there's some duplication, and then we can decide later if the different sections of code are similar enough that we can really extract a reusable component.

Adding the Single Post Route

Now that we have a `<SinglePostPage>` component, we can define a route to show it, and add links to each post in the front page feed.

We'll import `SinglePostPage` in `App.js`, and add the route:

`App.js`

JavaScript

```
import { PostsList } from './features/posts/PostsList'
import { AddPostForm } from './features/posts/AddPostForm'
import { SinglePostPage } from './features/posts/SinglePostPage'
```

```
function App() {
  return (
    <Router>
      <Navbar />
      <div className="App">
        <Switch>
          <Route
            exact
            path="/"
            render={() => (
              <React.Fragment>
                <AddPostForm />
                <PostsList />
              </React.Fragment>
            )}
          />
          <Route exact path="/posts/:postId"
component={SinglePostPage} />
          <Redirect to="/" />
        </Switch>
      </div>
    </Router>
  )
}
```

Then, in `<PostsList>`, we'll update the list rendering logic to include a `<Link>` that routes to that specific post.

`features/posts/PostsList.js`

JavaScript

```
import React from 'react'
import { useSelector } from 'react-redux'
import { Link } from 'react-router-dom'
```

```
export const PostsList = () => {
  const posts = useSelector(state => state.posts)

  const renderedPosts = posts.map(post => (
    <article className="post-excerpt" key={post.id}>
      <h3>{post.title}</h3>
      <p className="post-content">{post.content.substring(0,
100)}</p>
      <Link to={`/posts/${post.id}`} className="button
muted-button">
        View Post
      </Link>
    </article>
  ))

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

Since we can now navigate to another page, it would also be helpful to include a link back to the main posts page inside of the `<Navbar>` component:

app/Navbar.js

JavaScript

```
import React from 'react'

import { Link } from 'react-router-dom'

export const Navbar = () => {
  return (
    <nav>
```

```
      <section>
        <h1>Redux Essentials Example</h1>

        <div className="navContent">
          <div className="navLinks">
            <Link to="/">Posts</Link>
          </div>
        </div>
      </section>
    </nav>
  )
}
```

Now, you should be able to navigate to individual post pages and back to the main posts page with a few clicks. You can also link directly to a post by copying its URL, thanks to React Router.

Updating Post Entries

Next, we will add functionality for editing pre-existing posts. As a user, it's quite annoying to finish a task, save your progress, and then realize you made a mistake somewhere, so adding edit functionality when applicable is very useful for improving the user experience.

Let's add a new `<EditPostForm>` component that has the ability to take an existing post ID, read that post from the store, let the user edit the title and post content, and then save the changes to update the post in the store.

First, we need to update our `postsSlice` to create a new reducer function and an action so that the store knows how to actually update posts.

Inside of the `createSlice()` call, we should add a new function into the `reducers` object. Remember that the name of this reducer should be a good description of what's happening, because we're going to see the reducer name show up as part of the action type string in the Redux DevTools whenever this action is dispatched. Our first reducer was called `postAdded`, so let's call this one `postUpdated`.

In order to update a post object, we need to know:

- The ID of the post being updated, so that we can find the right post object in the state
- The new `title` and `content` fields that the user typed in

Redux action objects are required to have a `type` field, which is normally a descriptive string, and may also contain other fields with more information about what happened. By convention, we normally put the additional info in a field called `action.payload`, but it's up to us to decide what the `payload` field contains - it could be a string, a number, an object, an array, or something else. In this case, since we have three pieces of information we need, let's plan on having the `payload` field be an object with the three fields inside of it. That means the action object will look like `{type: 'posts/postUpdated', payload: {id, title, content}}`.

By default, the action creators generated by `createSlice` expect you to pass in one argument, and that value will be put into the action object as `action.payload`. So, we can pass an object containing those fields as the argument to the `postUpdated` action creator.

We also know that the reducer is responsible for determining how the state should actually be updated when an action is dispatched. Given that, we should have the reducer find the right post object based on the ID, and specifically update the `title` and `content` fields in that post.

Finally, we'll need to export the action creator function that `createSlice` generated for us, so that the UI can dispatch the new `postUpdated` action when the user saves the post.

Given all those requirements, here's how our `postsSlice` definition should look after we're done:

`features/posts/postsSlice.js`

JavaScript

```
const postsSlice = createSlice({
  name: 'posts',
  initialState,
  reducers: {
    postAdded(state, action) {
      state.push(action.payload)
    },
    postUpdated(state, action) {
      const { id, title, content } = action.payload
      const existingPost = state.find(post => post.id === id)
      if (existingPost) {
        existingPost.title = title
        existingPost.content = content
      }
    }
  }
```

```
})

export const { postAdded, postUpdated } = postsSlice.actions

export default postsSlice.reducer
```

The new `<EditPostForm>` component will look similar to the `<AddPostForm>`, but the logic needs to be a bit different. We need to retrieve the right `post` object from the store, then use that to initialize the state fields in the component so the user can make changes. We'll save the changed title and content values back to the store after the user is done. We'll also use React Router's history API to switch over to the single post page and show that post.

The code block for this new component spans the next couple of pages.

features/posts/EditPostForm.js

JavaScript

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

import { postUpdated } from './postsSlice'

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

  const post = useSelector(state =>
    state.posts.find(post => post.id === postId)
  )

  const [title, setTitle] = useState(post.title)
  const [content, setContent] = useState(post.content)

  const dispatch = useDispatch()
  const history = useHistory()

  const onTitleChanged = e => setTitle(e.target.value)
  const onContentChanged = e => setContent(e.target.value)
```

```
  const onSavePostClicked = () => {
    if (title && content) {
      dispatch(postUpdated({ id: postId, title, content }))
      history.push(`/posts/${postId}`)
    }
  }

  return (
    <section>
      <h2>Edit Post</h2>
      <form>
        <label htmlFor="postTitle">Post Title:</label>
        <input
          type="text"
          id="postTitle"
          name="postTitle"
          placeholder="What's on your mind?"
          value={title}
          onChange={onTitleChanged}
        />
        <label htmlFor="postContent">Content:</label>
        <textarea
          id="postContent"
          name="postContent"
          value={content}
          onChange={onContentChanged}
        />
      </form>
      <button type="button" onClick={onSavePostClicked}>
        Save Post
      </button>
    </section>
  )
}
```

Like with the `SinglePostPage`, we'll need to import `EditPostForm` into `App.js` and add a route that will render the component with the `postId` as a route parameter.

App.js

```
JavaScript
import { PostsList } from './features/posts/PostsList'
import { AddPostForm } from './features/posts/AddPostForm'
import { SinglePostPage } from './features/posts/SinglePostPage'
import { EditPostForm } from './features/posts/EditPostForm'

function App() {
  return (
    <Router>
      <Navbar />
      <div className="App">
        <Switch>
          <Route
            exact
            path="/"
            render={() => (
              <React.Fragment>
                <AddPostForm />
                <PostsList />
              </React.Fragment>
            )}
          />
          <Route exact path="/posts/:postId"
component={SinglePostPage} />
          <Route exact path="/editPost/:postId"
component={EditPostForm} />
          <Redirect to="/" />
        </Switch>
      </div>
    </Router>
  )
}
```

We should also add a new link to our `SinglePostPage` that will route to `EditPostForm`:

features/post/SinglePostPage.js

```javascript
import React from 'react'
import { useSelector } from 'react-redux'
import { Link } from 'react-router-dom'

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

  const post = useSelector(state =>
    state.posts.find(post => post.id === postId)
  )

  if (!post) {
    return (
      <section>
        <h2>Post not found!</h2>
      </section>
    )
  }

  return (
    <section>
      <article className="post">
        <h2>{post.title}</h2>
        <p className="post-content">{post.content}</p>
        <Link to={`/editPost/${post.id}`} className="button">
         Edit Post
       </Link>
      </article>
    </section>
  )
}
```

Preparing Action Payloads

We just saw that the action creators from `createSlice` normally expect one argument, which becomes `action.payload`. This simplifies the most common usage pattern, but sometimes we need to do more work to prepare the contents of an action object. In the case of our `postAdded` action, we need to generate a unique ID for the new post, and we also need to make sure that the payload is an object that looks like `{id, title, content}`.

Right now, we're generating the ID and creating the payload object in our React component, and passing the payload object into `postAdded`. But, what if we needed to dispatch the same action from different components, or the logic for preparing the payload is complicated? We'd have to duplicate that logic every time we wanted to dispatch the action, and we're forcing the component to know exactly what the payload for this action should look like.

Fortunately, `createSlice` lets us define a "prepare callback" function when we write a reducer. The "prepare callback" function can take multiple arguments, generate random values like unique IDs, and run whatever other synchronous logic is needed to decide what values go into the action object. It should then return an object with the `payload` field inside. The return object may also contain a `meta` field, which can be used to add extra descriptive values to the action, and an `error` field, which should be a boolean indicating whether this action represents some kind of an error.

Inside of the `reducers` field in `createSlice`, we can define one of the fields as an object that looks like `{reducer, prepare}`:

features/posts/postsSlice.js

JavaScript

```
const postsSlice = createSlice({
  name: 'posts',
  initialState,
  reducers: {
    postAdded: {
      reducer(state, action) {
        state.push(action.payload)
      },
      prepare(title, content) {
        return {
          payload: {
            id: nanoid(),
            title,
            content
          }
        }
      }
    }
    // other reducers here
  }
})
```

Now our component doesn't have to worry about what the payload object looks like - the action creator will take care of putting it together the right way. So, we can update the component so that it passes in `title` and `content` as arguments when it dispatches `postAdded`:

`features/posts/AddPostForm.js`

```
JavaScript
const onSavePostClicked = () => {
  if (title && content) {
    dispatch(postAdded(title, content))
    setTitle('')
    setContent('')
  }
}
```

This leaves our posts feature in a good place for now, but real applications often have many different features with many different slices of state. So far, we only have one.

You can’t have a social media app without getting other people involved, so let’s add the ability to keep track of a list of users in our app, and update the post-related functionality to make use of that data.

This new “users” feature will give us both a new feature and a new slice of state to make use of.

Adding a Users Slice

Since the concept of "users" is different from the concept of "posts", we want to keep the code and data for the users separated from the code and data for posts. We'll add a new `features/users` folder, and put a `usersSlice` file in there. Like with the posts slice, for now we'll add some initial entries so that we have data to work with.

`features/users/usersSlice.js`

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

const initialState = [
  { id: '0', name: 'Tianna Jenkins' },
  { id: '1', name: 'Kevin Grant' },
```

```
  { id: '2', name: 'Madison Price' }
]

const usersSlice = createSlice({
  name: 'users',
  initialState,
  reducers: {}
})

export default usersSlice.reducer
```

For now, we don't need to actually update the data, so we'll leave the `reducers` field as an empty object. We'll come back to this in a later section. As before, we'll import the `usersReducer` into our store file and add it to the store setup:

app/store.js

```
JavaScript
import { configureStore } from '@reduxjs/toolkit'
import postsReducer from '../features/posts/postsSlice'
import usersReducer from '../features/users/usersSlice'

export default configureStore({
  reducer: {
    posts: postsReducer,
    users: usersReducer
  }
})
```

Adding Authors for Posts

Every post in our app was written by one of our users, and every time we add a new post, we should keep track of which user wrote that post. In a real app, we'd have some sort of a `state.currentUser` field that keeps track of the current logged-in user, and use that information whenever they add a post.

To keep things simpler for this example, we'll update our `<AddPostForm>` component so that we can select a user from a dropdown list, and we'll include that user's ID as part of the post. Once

our post objects have a user ID in them, we can use that to look up the user's name and show it in each individual post in the UI.

First, we need to update our `postAdded` action creator to accept a user ID as an argument, and include that in the action. We'll also update the existing post entries in `initialState` to have a `post.user` field with one of the example user IDs.

features/posts/postsSlice.js

```javascript
const postsSlice = createSlice({
  name: 'posts',
  initialState,
  reducers: {
    postAdded: {
      reducer(state, action) {
        state.push(action.payload)
      },
      prepare(title, content, userId) {
        return {
          payload: {
            id: nanoid(),
            title,
            content,
            user: userId
          }
        }
      }
    }
    // other reducers
  }
})
```

Now, in our `<AddPostForm>`, we can read the list of users from the store with `useSelector` and show them as a dropdown. We'll then take the ID of the selected user and pass that to the `postAdded` action creator. While we're at it, we can add a bit of validation logic to our form so that the user can only click the "Save Post" button if the title and content inputs have some actual text in them:

features/posts/AddPostForm.js

JavaScript

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

import { postAdded } from './postsSlice'

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

  const dispatch = useDispatch()

  const users = useSelector(state => state.users)

  const onTitleChanged = e => setTitle(e.target.value)
  const onContentChanged = e => setContent(e.target.value)
  const onAuthorChanged = e => setUserId(e.target.value)

  const onSavePostClicked = () => {
    if (title && content) {
      dispatch(postAdded(title, content, userId))
      setTitle('')
      setContent('')
    }
  }

  const canSave = Boolean(title) && Boolean(content) &&
Boolean(userId)

  const usersOptions = users.map(user => (
    <option key={user.id} value={user.id}>
      {user.name}
    </option>
  ))

  return (
    <section>
      <h2>Add a New Post</h2>
      <form>
```

```
        <label htmlFor="postTitle">Post Title:</label>
        <input
          type="text"
          id="postTitle"
          name="postTitle"
          placeholder="What's on your mind?"
          value={title}
          onChange={onTitleChanged}
        />
        <label htmlFor="postAuthor">Author:</label>
        <select id="postAuthor" value={userId}
onChange={onAuthorChanged}>
          <option value=""></option>
          {usersOptions}
        </select>
        <label htmlFor="postContent">Content:</label>
        <textarea
          id="postContent"
          name="postContent"
          value={content}
          onChange={onContentChanged}
        />
        <button type="button" onClick={onSavePostClicked}
disabled={!canSave}>
          Save Post
        </button>
      </form>
    </section>
  )
}
```

Now, we need a way to show the name of the post's author inside of our post list items and <SinglePostPage>. Since we want to show this same kind of info in more than one place, we can make a PostAuthor component that takes a user ID as a prop, looks up the right user object, and formats the user's name:

features/posts/PostAuthor.js

```
JavaScript
import React from 'react'
import { useSelector } from 'react-redux'

export const PostAuthor = ({ userId }) => {
  const author = useSelector(state =>
    state.users.find(user => user.id === userId)
  )

  return <span>by {author ? author.name : 'Unknown
author'}</span>
}
```

Notice that we're following the same pattern in each of our components as we go. Any component that needs to read data from the Redux store can use the `useSelector` hook, and extract the specific pieces of data that it needs. Also, many components can access the same data in the Redux store at the same time.

We can now import the `PostAuthor` component into both `PostsList.js` and `SinglePostPage.js`, and render it as `<PostAuthor userId={post.user} />`, and every time we add a post entry, the selected user's name should show up inside of the rendered post.

Take a moment to accomplish those tasks before moving forward.

Storing Dates for Posts

Social media feeds are typically sorted by when the post was created, and show us the post creation time as a relative description like "5 hours ago". In order to do that, we need to start tracking a `date` field for our post entries.

Like with the `post.user` field, we'll update our `postAdded` prepare callback to make sure that `post.date` is always included when the action is dispatched. However, it's not another parameter that will be passed in. We want to always use the exact timestamp from when the action is dispatched, so we'll let the prepare callback handle that itself.

Since we can't just put a `Date` class instance into the Redux store, we'll track the `post.date` value as a timestamp string:

`features/posts/postsSlice.js`

```JavaScript
    postAdded: {
      reducer(state, action) {
        state.push(action.payload)
      },
      prepare(title, content, userId) {
        return {
          payload: {
            id: nanoid(),
            date: new Date().toISOString(),
            title,
            content,
            user: userId,
          },
        }
      },
    },
```

Like with post authors, we need to show the relative timestamp description in both our `<PostsList>` and `<SinglePostPage>` components. We'll add a `<TimeAgo>` component to handle formatting a timestamp string as a relative description. Libraries like `date-fns` have some useful utility functions for parsing and formatting dates, which we can use here:

`features/posts/TimeAgo.js`

```JavaScript
import React from 'react'
import { parseISO, formatDistanceToNow } from 'date-fns'

export const TimeAgo = ({ timestamp }) => {
  let timeAgo = ''
  if (timestamp) {
    const date = parseISO(timestamp)
    const timePeriod = formatDistanceToNow(date)
    timeAgo = `${timePeriod} ago`
  }

  return (
```

```
      <span title={timestamp}>
          <i>{timeAgo}</i>
      </span>
    )
  }
```

Sorting the Posts List

Our `<PostsList>` is currently showing all the posts in the same order the posts are kept in the Redux store. Our example has the oldest post first, and any time we add a new post, it gets added to the end of the posts array. That means the newest post is always at the bottom of the page.

Typically, social media feeds show the newest posts first, and you scroll down to see older posts. Even though the data is being kept oldest-first in the store, we can reorder the data in our `<PostsList>` component so that the newest post is first. In theory, since we know that the `state.posts` array is already sorted, we could just reverse the list. But, it's better to go ahead and sort it ourselves just to be sure.

Since `array.sort()` mutates the existing array, we need to make a copy of `state.posts` and sort that copy. We know that our `post.date` fields are being kept as date timestamp strings, and we can directly compare those to sort the posts in the right order.

`features/posts/PostsList.js`

JavaScript

```
// Sort posts in reverse chronological order by datetime string
const orderedPosts = posts.slice().sort((a, b) =>
b.date.localeCompare(a.date))

const renderedPosts = orderedPosts.map(post => {
  return (
    <article className="post-excerpt" key={post.id}>
      <h3>{post.title}</h3>
      <div>
        <PostAuthor userId={post.user} />
        <TimeAgo timestamp={post.date} />
      </div>
```

```
      <p className="post-content">{post.content.substring(0,
100)}</p>
      <Link to={`/posts/${post.id}`} className="button
muted-button">
        View Post
      </Link>
    </article>
  )
})
```

We also need to add the `date` field to `initialState` in `postsSlice.js`. We'll use `date-fns` here again to subtract minutes from the current date/time so they differ from each other.

features/posts/postsSlice.js

JavaScript

```
import { createSlice, nanoid } from '@reduxjs/toolkit'
import { sub } from 'date-fns'

const initialState = [
  {
    // omitted fields
    content: 'Hello!',
    date: sub(new Date(), { minutes: 10 }).toISOString()
  },
  {
    // omitted fields
    content: 'More text',
    date: sub(new Date(), { minutes: 5 }).toISOString()
  }
]
```

Post Reaction Buttons

We have one more new feature to add for this section.

We'll add a row of emoji reaction buttons at the bottom of each post in `<PostsList>` and `<SinglePostPage>`. Every time a user clicks one of the reaction buttons, we'll need to update a

matching counter field for that post in the Redux store. Since the reaction counter data is in the Redux store, switching between different parts of the app should consistently show the same values in any component that uses that data.

Like with post authors and timestamps, we want to use this everywhere we show posts, so we'll create a `<ReactionButtons>` component that takes a `post` as a prop. We'll start by just showing the buttons inside, with the current reaction counts for each button:

`features/posts/ReactionButtons.js`

```
JavaScript
import React from 'react'

const reactionEmoji = {
  thumbsUp: '👍',
  hooray: '🎉',
  heart: '❤️',
  rocket: '🚀',
  eyes: '👀'
}

export const ReactionButtons = ({ post }) => {
  const reactionButtons = Object.entries(reactionEmoji).map(([name,
emoji]) => {
    return (
      <button key={name} type="button" className="muted-button
reaction-button">
        {emoji} {post.reactions[name]}
      </button>
    )
  })

  return <div>{reactionButtons}</div>
}
```

We don't yet have a `post.reactions` field in our data, so we'll need to update the `initialState` post objects and our `postAdded` prepare callback function to make sure that every post has that data inside, like `reactions: {thumbsUp: 0, hooray: 0, heart: 0, rocket: 0, eyes: 0}`. Take a moment to do so before continuing.

Now, we can define a new reducer that will handle updating the reaction count for a post when a user clicks the reaction button.

Like with editing posts, we need to know the ID of the post, and which reaction button the user clicked on. We'll have our `action.payload` be an object that looks like `{id, reaction}`. The reducer can then find the right post object, and update the correct reactions field.

`features/posts/postsSlice.js`

```javascript
const postsSlice = createSlice({
  name: 'posts',
  initialState,
  reducers: {
    reactionAdded(state, action) {
      const { postId, reaction } = action.payload
      const existingPost = state.find(post => post.id === postId)
      if (existingPost) {
        existingPost.reactions[reaction]++
      }
    }
    // other reducers
  }
})

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

As we've seen already, `createSlice` lets us write "mutating" logic in our reducers. If we weren't using `createSlice` and the Immer library, the line `existingPost.reactions[reaction]++` would indeed mutate the existing `post.reactions` object, and this would probably cause bugs elsewhere in our app because we didn't follow the rules of reducers. Since we *are* using `createSlice`, we can write this more complex update logic in a simpler way, and let Immer do the work of turning this code into a safe immutable update.

Notice that **our action object just contains the minimum amount of information needed to describe what happened**. We know which post we need to update, and which reaction name was clicked on. We *could* have calculated the new reaction counter value and put that in the action, but **it's always better to keep the action objects as small as possible, and do the state update calculations in the reducer**. This also means that reducers can contain as much logic as necessary to calculate the new state.

Our last step is to update the `<ReactionButtons>` component to dispatch the `reactionAdded` action when the user clicks a button:

`features/posts/ReactionButtons.js`

```JavaScript
import React from 'react'
import { useDispatch } from 'react-redux'

import { reactionAdded } from './postsSlice'

const reactionEmoji = {
  thumbsUp: '👍',
  hooray: '🎉',
  heart: '❤️',
  rocket: '🚀',
  eyes: '👀'
}

export const ReactionButtons = ({ post }) => {
  const dispatch = useDispatch()

  const reactionButtons = Object.entries(reactionEmoji).map(([name,
emoji]) => {
    return (
      <button
        key={name}
        type="button"
        className="muted-button reaction-button"
        onClick={() =>
          dispatch(reactionAdded({ postId: post.id, reaction: name }))
        }
      >
        {emoji} {post.reactions[name]}
      </button>
    )
  })

  return <div>{reactionButtons}</div>
}
```

If you haven't already done so, add your new `<ReactionButtons>` component to your `<PostsList>` and make sure the buttons are working properly. You can explore your actions and how the components respond to them via the Redux DevTools, and take a moment to address any errors that may have occurred during this part of the process.

Part Two Recap

There are some important things to remember from this section:

- **Any React component can use data from the Redux store as needed.**
  - Any component can read any data that is in the Redux store.
  - Multiple components can read the same data, even at the same time.
  - Components should extract the smallest amount of data they need to render themselves.
  - Components can combine values from props, state, and the Redux store to determine what UI they need to render. They can read multiple pieces of data from the store, and reshape the data as needed for display.
  - Any component can dispatch actions to cause state updates.
- **Redux action creators can prepare action objects with the right contents.**
  - `createSlice` and `createAction` can accept a "prepare callback" that returns the action payload.
  - Unique IDs and other random values should be put in the action, not calculated in the reducer.
- **Reducers should contain the actual state update logic.**
  - Reducers can contain whatever logic is needed to calculate the next state.
  - Action objects should contain just enough info to describe what happened.

Here's what your application should look like so far: Part Two Completed CodeSandbox.

Part Three Preview

So far, we've only been using data that has existed in our initial state or that was added by the user. In the next section, we'll explore adding external data to the application through use of a server API.

Part 3: Data Fetching

Our application currently uses a local state that only updates with interaction from the user. In the case of almost every web application, the application's front-end will need to work with some server back-end through the use of an API.

In this part of this lab, we'll convert our application to fetch the posts and users data from an API, and add new posts by saving them to the API.

To keep this project isolated but realistic, the initial setup already includes a fake in-memory REST API for our data. 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 fake REST API for this section.

Thunks and Async Logic

In this section, we'll be using the `createAsyncThunk` API from RTK. If you are unfamiliar with thunks or their usage in Redux, take some time to review the Redux: Writing Logic with Thunks usage guide page.

Loading Posts

So far, our `postsSlice` has used some hard-coded 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.

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`

```
JavaScript
const postsSlice = createSlice(/* omitted 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`

```
JavaScript
// omitted imports
import { selectAllPosts } from './postsSlice'

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

`features/posts/SinglePostPage.js`

```
JavaScript
// omitted imports
import { selectPostById } from './postsSlice'

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

  const post = useSelector(state => selectPostById(state, postId))
```

```
  // omitted component logic
}
```

features/posts/EditPostForm.js

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

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

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

It's often a good idea to encapsulate data lookups by writing reusable selectors, as above. You can also create "memoized" selectors that can help improve performance, which we'll look at in the next part of this project.

However, like any abstraction, this is 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:

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

These fields would exist alongside whatever actual data is being stored.

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`

```
JavaScript
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)
```

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 our API at `'/fakeApi/posts'`.

features/posts/postsSlice.js

```JavaScript
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
})
```

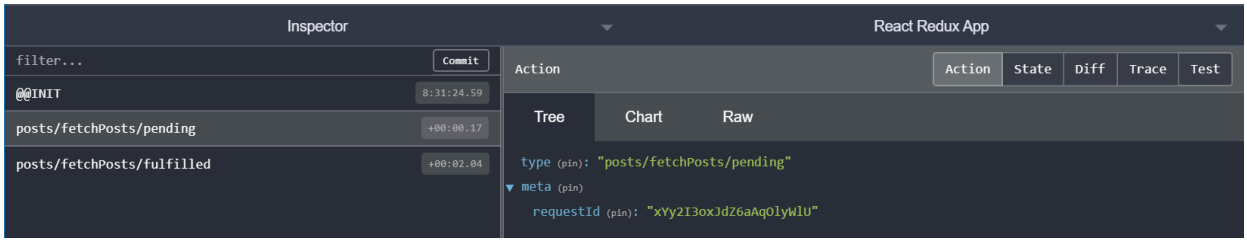
`createAsyncThunk` accepts two arguments:

- 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.

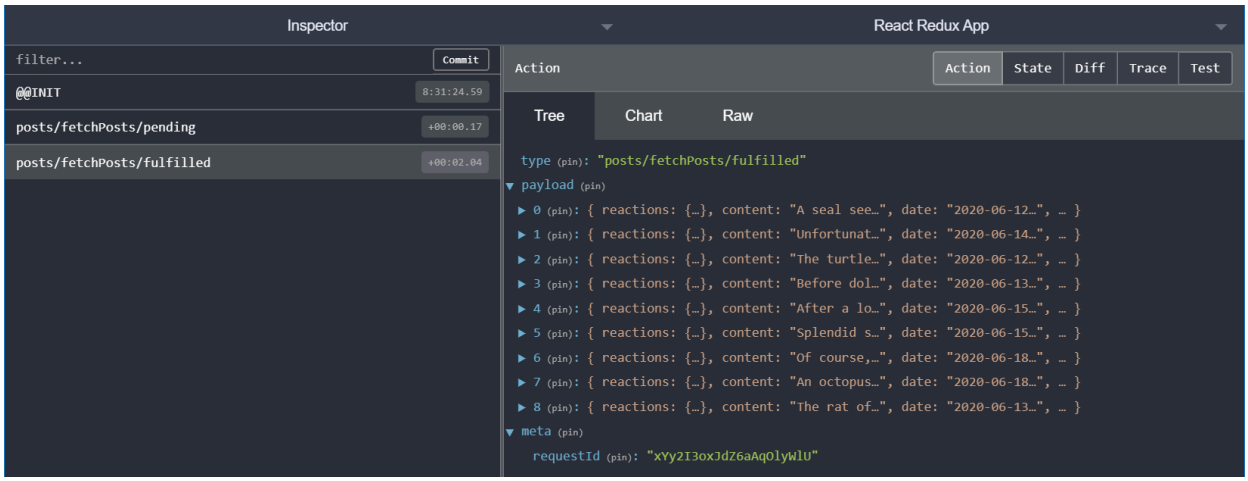
In this case, we pass in `'posts/fetchPosts'` as the action type prefix. Our payload creation callback waits for the API call to return a response. The response object looks like `{data: []}`, and we want our dispatched Redux action to have a payload that is just the array of posts. So, we extract `response.data`,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

Now, we can update our `<PostsList>` component to actually fetch this data automatically.

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`

JavaScript

```
import React, { useEffect } from 'react'
import { useSelector, useDispatch } from 'react-redux'
// omitted 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])

  // omitted 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

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. 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.

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:

features/posts/postsSlice.js

```javascript
const postsSlice = createSlice({
  name: 'posts',
  initialState,
  reducers: { /* omitted existing reducers here */ },
  extraReducers(builder) {
    builder
      .addCase(fetchPosts.pending, (state, action) => {
        state.status = 'loading'
      })
      .addCase(fetchPosts.fulfilled, (state, action) => {
        state.status = 'succeeded'
        state.posts = state.posts.concat(action.payload)
      })
      .addCase(fetchPosts.rejected, (state, action) => {
        state.status = 'failed'
        state.error = action.error.message
      })
  }
})
```

This handles 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 re-rendering 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`

JavaScript

```
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>
  }

  // omitted return
```

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`

JavaScript

```
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.

In this case, the initial state was an empty array, and we probably could have done `state.push(...action.payload)` to mutate it. 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.

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`

JavaScript

```
// omitted 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()
```

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.

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

```
JavaScript
export const addNewPost = createAsyncThunk(
  'posts/addNewPost',
  // Payload creator receives 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, including unique ID
    return response.data
  }
)

const postsSlice = createSlice({
  name: 'posts',
  initialState,
  reducers: {
    // The existing `postAdded` reducer & prepare callback were deleted
    reactionAdded(state, action) {}, // omitted logic
    postUpdated(state, action) {} // omitted logic
  },
  extraReducers(builder) {
    // omitted 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`

JavaScript

```
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')

  // omitted 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')
      }
    }
  }
```

```
  // omitted 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. However, 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. 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.

Part Three Recap

Here's what we discovered in this section:

- **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.

Here's what your application should look like so far: Part Three Completed CodeSandbox.

Part 4: Performance Optimizations and Normalizing Data

In this section, we'll look at optimized patterns for ensuring good performance in our application, and techniques for automatically handling common updates of data in the store.

So far, most of our functionality has been centered around the `posts` feature. We're going to add a couple new sections of the app. After those are added, we'll look at some specific details of how we've built things, and talk about some weaknesses with what we've built so far and how we can improve the implementation.

Adding User Pages

We're fetching a list of users from our fake API, and we can choose a user as the author when we add a new post. But, a social media app needs the ability to look at the page for a specific user and see all the posts they've made. Let's add a page to show the list of all users, and another to show all posts by a specific user.

We'll start by adding a new `<UsersList>` component. It follows the usual pattern of reading some data from the store with `useSelector`, and mapping over the array to show a list of users with links to their individual pages:

features/users/UsersList.js

```JavaScript
import React from 'react'
import { useSelector } from 'react-redux'
import { Link } from 'react-router-dom'
import { selectAllUsers } from './usersSlice'

export const UsersList = () => {
  const users = useSelector(selectAllUsers)

  const renderedUsers = users.map(user => (
    <li key={user.id}>
      <Link to={`/users/${user.id}`}>{user.name}</Link>
    </li>
  ))

  return (
    <section>
      <h2>Users</h2>

      <ul>{renderedUsers}</ul>
    </section>
  )
}
```

We don't yet have a `selectAllUsers` selector, so we'll need to add that to `usersSlice.js` along with a `selectUserById` selector:

features/users/usersSlice.js

```JavaScript
export default usersSlice.reducer

export const selectAllUsers = state => state.users

export const selectUserById = (state, userId) =>
  state.users.find(user => user.id === userId)
```

And we'll add a `<UserPage>`, which is similar to our `<SinglePostPage>` in taking a `userId` parameter from the router:

`features/users/UserPage.js`

```
JavaScript
import React from 'react'
import { useSelector } from 'react-redux'
import { Link } from 'react-router-dom'

import { selectUserById } from '../users/usersSlice'
import { selectAllPosts } from '../posts/postsSlice'

export const UserPage = ({ match }) => {
  const { userId } = match.params

  const user = useSelector(state => selectUserById(state,
userId))

  const postsForUser = useSelector(state => {
    const allPosts = selectAllPosts(state)
    return allPosts.filter(post => post.user === userId)
  })

  const postTitles = postsForUser.map(post => (
    <li key={post.id}>
      <Link to={`/posts/${post.id}`}>{post.title}</Link>
    </li>
  ))

  return (
    <section>
      <h2>{user.name}</h2>

      <ul>{postTitles}</ul>
    </section>
  )
}
```

As we've seen before, we can take data from one `useSelector` call, or from props, and use that to help decide what to read from the store in another `useSelector` call.

As usual, we will add routes for these components in `<App>`:

App.js

```
JavaScript
<Route exact path="/posts/:postId" component={SinglePostPage} />
<Route exact path="/editPost/:postId" component={EditPostForm} />
<Route exact path="/users" component={UsersList} />
<Route exact path="/users/:userId" component={UserPage} />
<Redirect to="/" />
```

We'll also add another tab in `<Navbar>` that links to `/users` so that we can click and go to `<UsersList>`:

app/Navbar.js

```
JavaScript
export const Navbar = () => {
  return (
    <nav>
      <section>
        <h1>Redux Essentials Example</h1>

        <div className="navContent">
          <div className="navLinks">
            <Link to="/">Posts</Link>
            <Link to="/users">Users</Link>
          </div>
        </div>
      </section>
    </nav>
  )
}
```

Adding Notifications

No social media app would be complete without some notifications popping up to tell us that someone has sent a message, left a comment, or reacted to one of our posts.

In a real application, our app client would be in constant communication with the backend server, and the server would push an update to the client every time something happens. Since this is a small example app, we're going to mimic that process by adding a button to actually fetch some notification entries from our fake API. We also don't have any other real users sending messages or reacting to posts, so the fake API will just create some random notification entries every time we make a request. (Remember, the goal here is to see how to use Redux itself.)

Notifications Slice

Since this is a new part of our app, the first step is to create a new slice for our notifications, and an async thunk to fetch some notification entries from the API. In order to create some realistic notifications, we'll include the timestamp of the latest notification we have in state. That will let our mock server generate notifications newer than that timestamp.

`features/notifications/notificationsSlice.js`

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

export const fetchNotifications = createAsyncThunk(
  'notifications/fetchNotifications',
  async (_, { getState }) => {
    const allNotifications = selectAllNotifications(getState())
    const [latestNotification] = allNotifications
    const latestTimestamp = latestNotification ?
latestNotification.date : ''
    const response = await client.get(
      `/fakeApi/notifications?since=${latestTimestamp}`
    )
    return response.data
  }
)

const notificationsSlice = createSlice({
  name: 'notifications',
```

```
  initialState: [],
  reducers: {},
  extraReducers(builder) {
    builder.addCase(fetchNotifications.fulfilled, (state, action)
=> {
      state.push(...action.payload)
      // Sort with newest first
      state.sort((a, b) => b.date.localeCompare(a.date))
    })
  }
})

export default notificationsSlice.reducer
export const selectAllNotifications = state =>
state.notifications
```

As with the other slices, import `notificationsReducer` into `store.js` and add it to the `configureStore()` call.

We've written an async thunk called `fetchNotifications`, which will retrieve a list of new notifications from the server. As part of that, we want to use the creation timestamp of the most recent notification as part of our request, so that the server knows it should only send back notifications that are actually new.

We know that we will be getting back an array of notifications, so we can pass them as separate arguments to `state.push()`, and the array will add each item. We also want to make sure that they're sorted so that the most recent notification is first in the array, just in case the server were to send them out of order. (As a reminder, `array.sort()` always mutates the existing array - this is only safe because we're using `createSlice` and Immer inside.)

Thunk Arguments

If you look at our `fetchNotifications` thunk, it has something new that we haven't seen before. Let's talk about thunk arguments for a minute.

We've already seen that we can pass an argument into a thunk action creator when we dispatch it, like `dispatch(addPost(newPost))`. For `createAsyncThunk` specifically, you can only pass in one argument, and whatever we pass in becomes the first argument of the payload creation callback.

The second argument to our payload creator is a `thunkAPI` object containing several useful functions and pieces of information:

- `dispatch` and `getState`: the actual `dispatch` and `getState` methods from our Redux store. You can use these inside the thunk to dispatch more actions, or get the latest Redux store state (such as reading an updated value after another action is dispatched).
- `extra`: the "extra argument" that can be passed into the thunk middleware when creating the store. This is typically some kind of API wrapper, such as a set of functions that know how to make API calls to your application's server and return data, so that your thunks don't have to have all the URLs and query logic directly inside.
- `requestId`: a unique random ID value for this thunk call. Useful for tracking status of an individual request.
- `signal`: An `AbortController.signal` function that can be used to cancel an in-progress request.
- `rejectWithValue`: a utility that helps customize the contents of a rejected action if the thunk receives an error.

If you're writing a thunk by hand instead of using `createAsyncThunk`, the thunk function will get (`dispatch`, `getState`) as separate arguments, instead of putting them together in one object.

In this case, we know that the list of notifications is in our Redux store state, and that the latest notification should be first in the array. We can destructure the `getState` function out of the `thunkAPI` object, call it to read the state value, and use the `selectAllNotifications` selector to give us just the array of notifications. Since the array of notifications is sorted newest first, we can grab the latest one using array destructuring.

Adding the Notifications List

With that slice created, we can add a `<NotificationsList>` component:

`features/notifications/NotificationsList.js`

JavaScript

```
import React from 'react'
import { useSelector } from 'react-redux'
import { formatDistanceToNow, parseISO } from 'date-fns'

import { selectAllUsers } from '../users/usersSlice'

import { selectAllNotifications } from './notificationsSlice'
```

```
export const NotificationsList = () => {
  const notifications = useSelector(selectAllNotifications)
  const users = useSelector(selectAllUsers)

  const renderedNotifications = notifications.map(notification =>
{
    const date = parseISO(notification.date)
    const timeAgo = formatDistanceToNow(date)
    const user = users.find(user => user.id ===
notification.user) || {
      name: 'Unknown User'
    }

    return (
      <div key={notification.id} className="notification">
        <div>
          <b>{user.name}</b> {notification.message}
        </div>
        <div title={notification.date}>
          <i>{timeAgo} ago</i>
        </div>
      </div>
    )
  })

  return (
    <section className="notificationsList">
      <h2>Notifications</h2>
      {renderedNotifications}
    </section>
  )
}
```

Once again, we're reading a list of items from the Redux state, mapping over them, and rendering content for each item.

We also need to update the `<Navbar>` to add a "Notifications" tab, and a new button to fetch some notifications:

app/Navbar.js

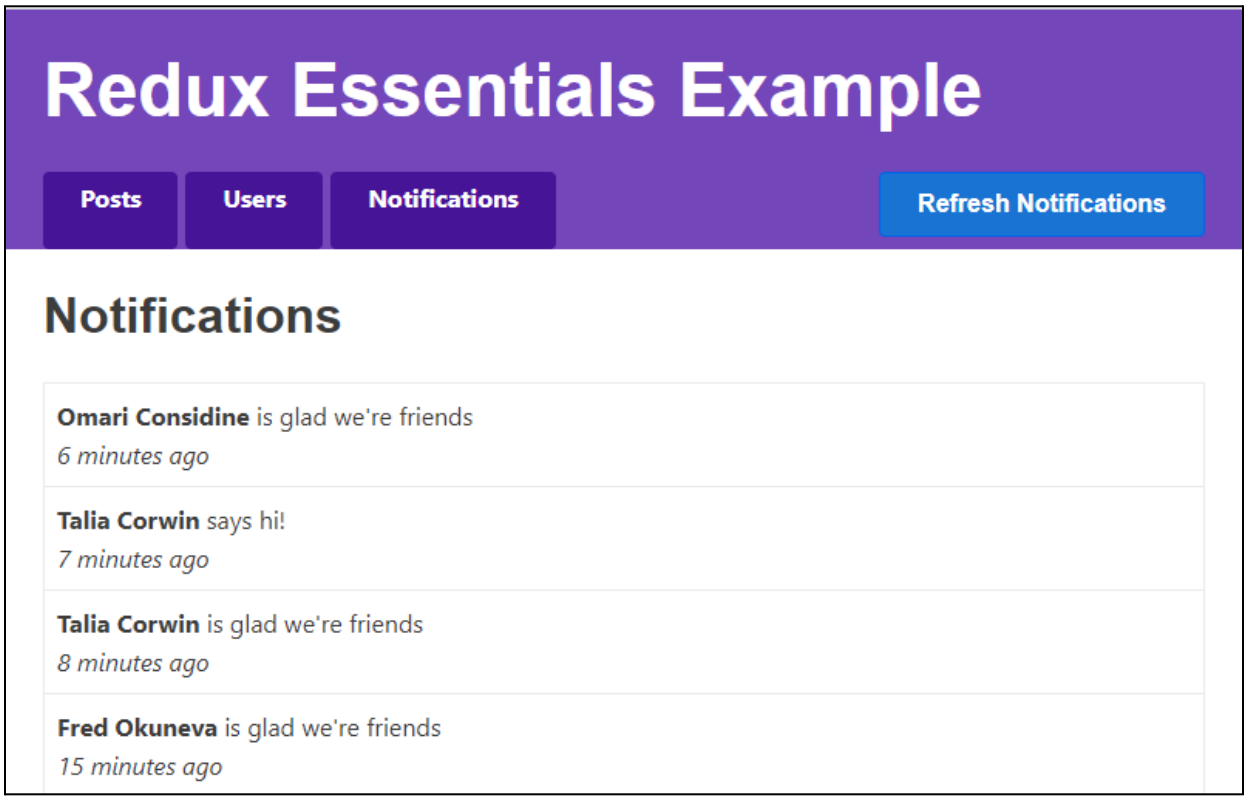
```JavaScript
import React from 'react'
import { useDispatch } from 'react-redux'
import { Link } from 'react-router-dom'

import { fetchNotifications } from
'../features/notifications/notificationsSlice'

export const Navbar = () => {
  const dispatch = useDispatch()

  const fetchNewNotifications = () => {
    dispatch(fetchNotifications())
  }

  return (
    <nav>
      <section>
        <h1>Redux Essentials Example</h1>

        <div className="navContent">
          <div className="navLinks">
            <Link to="/">Posts</Link>
            <Link to="/users">Users</Link>
            <Link to="/notifications">Notifications</Link>
          </div>
          <button className="button"
onClick={fetchNewNotifications}>
            Refresh Notifications
          </button>
        </div>
      </section>
    </nav>
  )
}
```

Lastly, we need to update `App.js` with the "Notifications" route so we can navigate to it:

App.js

```JavaScript
// omitted imports
import { NotificationsList } from
'./features/notifications/NotificationsList'

function App() {
  return (
    <Router>
      <Navbar />
      <div className="App">
        <Switch>
          <Route exact path="/notifications"
component={NotificationsList} />
          // omitted existing routes
          <Redirect to="/" />
        </Switch>
      </div>
    </Router>
  )
}
```

Here's what the "Notifications" tab looks like so far:



Showing New Notifications

Each time we click "Refresh Notifications", a few more notification entries will be added to our list. In a real app, those could be coming from the server while we're looking at other parts of the UI. We can do something similar by clicking "Refresh Notifications" while we're looking at the `<PostsList>` or `<UserPage>`. But, right now we have no idea how many notifications just arrived, and if we keep clicking the button, there could be many notifications we haven't read yet. Let's add some logic to keep track of which notifications have been read and which of them are "new". That will let us show the count of "Unread" notifications as a badge on our "Notifications" tab in the navbar, and display new notifications in a different color.

Our fake API is already sending back the notification entries with `isNew` and `read` fields, so we can use those in our code.

First, we'll update `notificationsSlice` to have a reducer that marks all notifications as read, and some logic to handle marking existing notifications as "not new":

`features/notifications/notificationsSlide.js`

```JavaScript
const notificationsSlice = createSlice({
  name: 'notifications',
  initialState: [],
  reducers: {
    allNotificationsRead(state, action) {
      state.forEach(notification => {
        notification.read = true
      })
    }
  },
  extraReducers(builder) {
    builder.addCase(fetchNotifications.fulfilled, (state, action)
=> {
      state.push(...action.payload)
      state.forEach(notification => {
        // Any notifications we've read are no longer new
        notification.isNew = !notification.read
      })
      // Sort with newest first
      state.sort((a, b) => b.date.localeCompare(a.date))
    })
  }
})

export const { allNotificationsRead } =
notificationsSlice.actions

export default notificationsSlice.reducer
```

We want to mark these notifications as read whenever our `<NotificationsList>` component renders, either because we clicked on the tab to view the notifications, or because we already have it open and we just received some additional notifications. We can do this by dispatching `allNotificationsRead` any time this component re-renders. In order to avoid flashing of old data as this updates, we'll dispatch the action in a `useLayoutEffect` hook. We also want to add an additional class name to any notification list entries in the page, to highlight them:

`features/notifications/NotificationsList.js`

JavaScript

```
import React, { useLayoutEffect } from 'react'
import { useSelector, useDispatch } from 'react-redux'
import { formatDistanceToNow, parseISO } from 'date-fns'
import classnames from 'classnames'

import { selectAllUsers } from '../users/usersSlice'

import {
  selectAllNotifications,
  allNotificationsRead
} from './notificationsSlice'

export const NotificationsList = () => {
  const dispatch = useDispatch()
  const notifications = useSelector(selectAllNotifications)
  const users = useSelector(selectAllUsers)

  useLayoutEffect(() => {
    dispatch(allNotificationsRead())
  })

  const renderedNotifications = notifications.map(notification =>
{
    const date = parseISO(notification.date)
    const timeAgo = formatDistanceToNow(date)
    const user = users.find(user => user.id ===
notification.user) || {
      name: 'Unknown User'
    }

    const notificationClassname = classnames('notification', {
      new: notification.isNew
    })

    return (
      <div key={notification.id}
className={notificationClassname}>
        <div>
          <b>{user.name}</b> {notification.message}
```

```
          </div>
          <div title={notification.date}>
            <i>{timeAgo} ago</i>
          </div>
        </div>
      )
    })

    return (
      <section className="notificationsList">
        <h2>Notifications</h2>
        {renderedNotifications}
      </section>
    )
  }
```

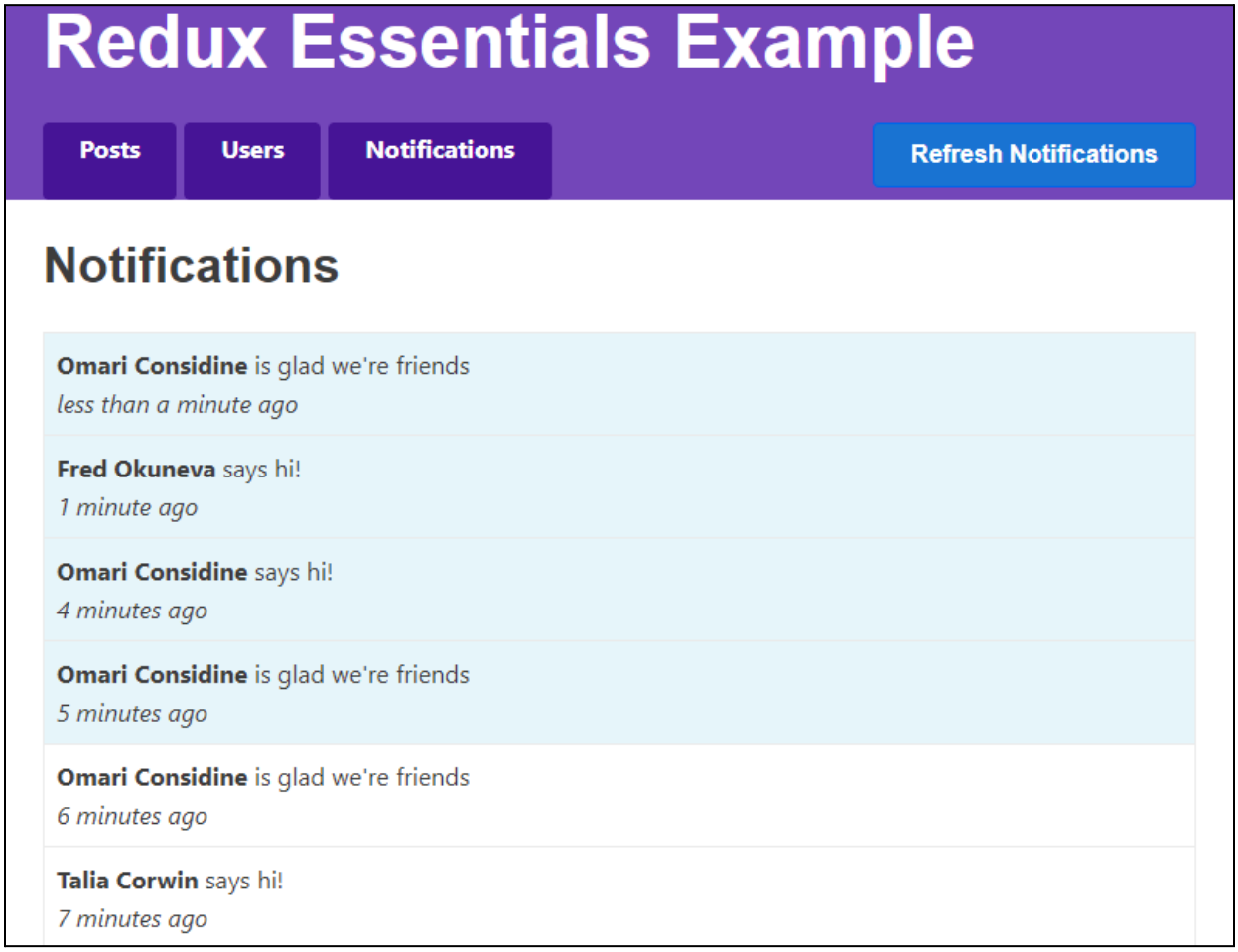
This works, but actually has a slightly surprising bit of behavior. Any time there are new notifications (either because we've just switched to this tab, or we've fetched some new notifications from the API), you'll actually see two `"notifications/allNotificationsRead"` actions dispatched. Why is that?

Let's say we have fetched some notifications while looking at the `<PostsList>`, and then click the "Notifications" tab. The `<NotificationsList>` component will mount, and the `useLayoutEffect` callback will run after that first render and dispatch `allNotificationsRead`. Our `notificationsSlice` will handle that by updating the notification entries in the store. This creates a new `state.notifications` array containing the immutably-updated entries, which forces our component to render again because it sees a new array returned from the `useSelector`, and the `useLayoutEffect` hook runs again and dispatches `allNotificationsRead` a second time. The reducer runs again, but this time no data changes, so the component doesn't re-render.

There is a couple ways we could potentially avoid that second dispatch, like splitting the logic to dispatch once when the component mounts, and only dispatch again if the size of the notifications array changes. But, this isn't actually hurting anything, so we can leave it alone.

This does actually show that **it's possible to dispatch an action and not have *any* state changes happen at all**. Remember, **it's always up to your reducers to decide *if* any state actually needs to be updated, and "nothing needs to happen" is a valid decision for a reducer to make**.

Here's how the notifications tab looks now that we've got the "new/read" behavior working:



The last thing we need to do before we move on is to add the badge on our "Notifications" tab in the navbar. This will show us the count of "Unread" notifications when we are in other tabs:

app/Navbar.js

JavaScript

```
// omitted imports
import { useDispatch, useSelector } from 'react-redux'

import {
  fetchNotifications,
  selectAllNotifications
} from '../features/notifications/notificationsSlice'
```

```
export const Navbar = () => {
  const dispatch = useDispatch()
  const notifications = useSelector(selectAllNotifications)
  const numUnreadNotifications = notifications.filter(n =>
!n.read).length
  // omitted component contents
  let unreadNotificationsBadge

  if (numUnreadNotifications > 0) {
    unreadNotificationsBadge = (
      <span className="badge">{numUnreadNotifications}</span>
    )
  }
  return (
    <nav>
      // omitted component contents
      <div className="navLinks">
        <Link to="/">Posts</Link>
        <Link to="/users">Users</Link>
        <Link to="/notifications">
          Notifications {unreadNotificationsBadge}
        </Link>
      </div>
      // omitted component contents
    </nav>
  )
}
```

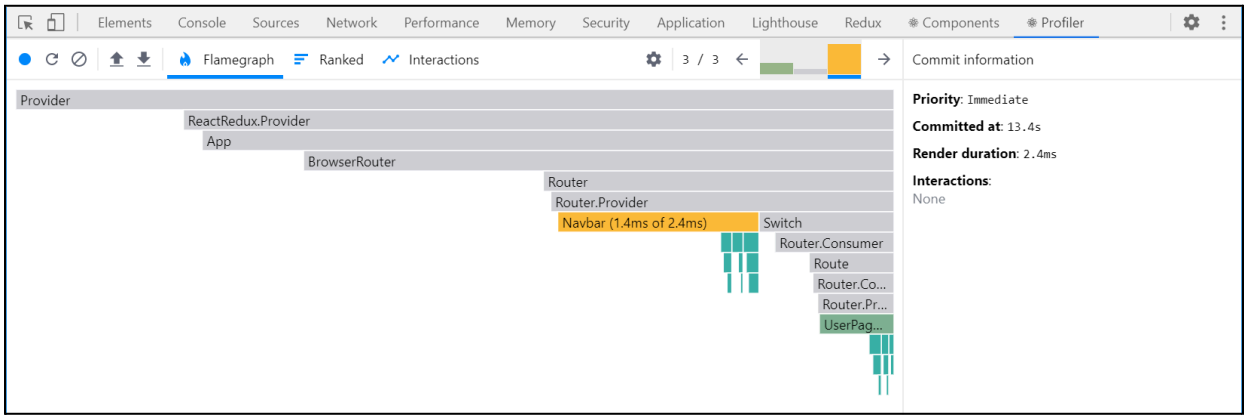
Improving Render Performance

Our application is looking useful, but we've actually got a couple flaws in when and how our components re-render. Let's look at those problems, and talk about some ways to improve the performance.

Investigating Render Behavior

We can use the React DevTools Profiler to view some graphs of what components re-render when state is updated. Try clicking over to the `<UserPage>` for a single user. Open up your browser's DevTools, and in the React "Profiler" tab, click the circle "Record" button in the

upper-left. Then, click the "Refresh Notifications" button in our app, and stop the recording in the React DevTools Profiler. You should see a chart that looks like this:



We can see that the `<Navbar>` re-rendered, which makes sense because it had to show the updated "unread notifications" badge in the tab. But, why did our `<UserPage>` re-render?

If we inspect the last couple dispatched actions in the Redux DevTools, we can see that only the notifications state updated. Since the `<UserPage>` doesn't read any notifications, it shouldn't have re-rendered. Something must be wrong with the component.

If we look at `<UserPage>` carefully, there's a specific problem:

`features/UserPage.js`

JavaScript

```
export const UserPage = ({ match }) => {
  const { userId } = match.params

  const user = useSelector(state => selectUserById(state,
userId))

  const postsForUser = useSelector(state => {
    const allPosts = selectAllPosts(state)
    return allPosts.filter(post => post.user === userId)
  })

  // omitted rendering logic
}
```

We know that `useSelector` will re-run every time an action is dispatched, and that it forces the component to re-render if we return a new reference value.

We're calling `filter()` inside of our `useSelector` hook, so that we only return the list of posts that belong to this user. Unfortunately, **this means that `useSelector` always returns a new array reference, and so our component will re-render after every action even if the posts data hasn't changed!**

Memoizing Selector Functions

What we really need is a way to only calculate the new filtered array if either `state.posts` or `userId` have changed. If they haven't changed, we want to return the same filtered array reference as the last time.

This idea is called "memoization". We want to save a previous set of inputs and the calculated result, and if the inputs are the same, return the previous result instead of recalculating it again.

So far, we've been writing selector functions by ourselves, and just so that we don't have to copy and paste the code for reading data from the store. It would be great if there was a way to make our selector functions memoized.

Reselect is a library for creating memoized selector functions, and was specifically designed to be used with Redux. It has a `createSelector` function that generates memoized selectors that will only recalculate results when the inputs change. Redux Toolkit exports the `createSelector` function, so we already have it available.

Let's make a new `selectPostsByUser` selector function, using Reselect, and use it here.

`features/posts/postsSlice.js`

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

// omitted slice logic

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

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

export const selectPostsByUser = createSelector(
```

```
  [selectAllPosts, (state, userId) => userId],
  (posts, userId) => posts.filter(post => post.user === userId)
)
```

`createSelector` takes one or more "input selector" functions as arguments, plus an "output selector" function. When we call `selectPostsByUser(state, userId)`, `createSelector` will pass all of the arguments into each of our input selectors. Whatever those input selectors return becomes the arguments for the output selector.

In this case, we know that we need the array of all posts and the user ID as the two arguments for our output selector. We can reuse our existing `selectAllPosts` selector to extract the posts array. Since the user ID is the second argument we're passing into `selectPostsByUser`, we can write a small selector that just returns `userId`.

Our output selector then takes `posts` and `userId`, and returns the filtered array of posts for just that user.

If we try calling `selectPostsByUser` multiple times, it will only re-run the output selector if either `posts` or `userId` has changed. For example:

JavaScript

```
const state1 = getState()
// Output selector runs, because it's the first call
selectPostsByUser(state1, 'user1')
// Output selector does _not_ run, because the arguments haven't
changed
selectPostsByUser(state1, 'user1')
// Output selector runs, because `userId` changed
selectPostsByUser(state1, 'user2')

dispatch(reactionAdded())
const state2 = getState()
// Output selector does not run, because `posts` and `userId` are
the same
selectPostsByUser(state2, 'user2')

// Add some more posts
dispatch(addNewPost())
```

```
const state3 = getState()
// Output selector runs, because `posts` has changed
selectPostsByUser(state3, 'user2')
```

If we call this selector in `<UserPage>` and re-run the React profiler while fetching notifications, we should see that `<UserPage>` doesn't re-render this time:

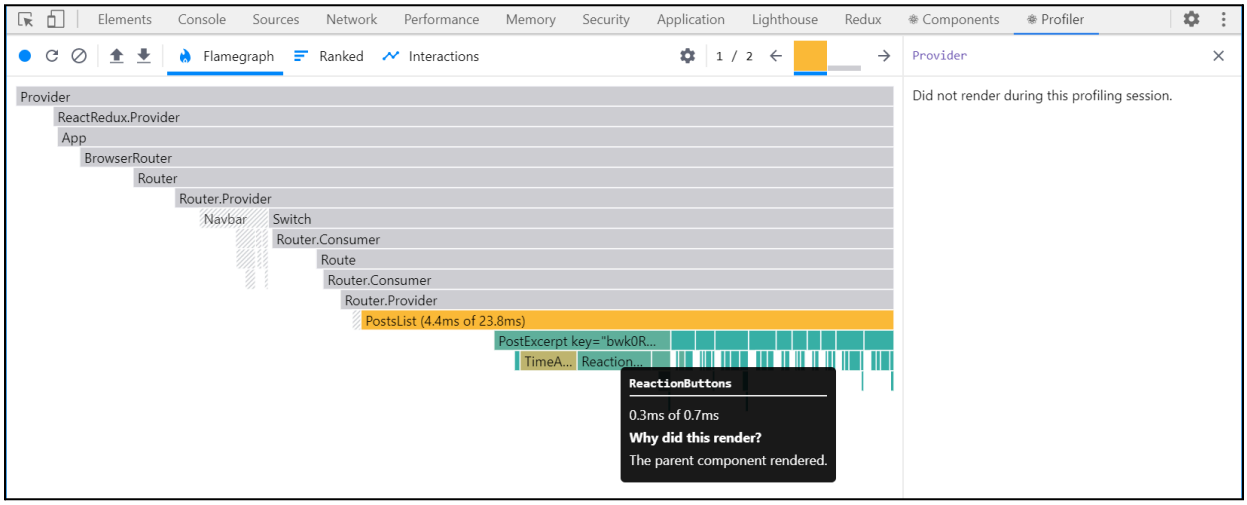
features/users/UserPage.js

```
JavaScript
export const UserPage = ({ match }) => {
  const { userId } = match.params

  const user = useSelector(state => selectUserById(state,
userId))

  const postsForUser = useSelector(state =>
selectPostsByUser(state, userId))

  // omitted rendering logic
}
```

Memoized selectors are a valuable tool for improving performance in a React-Redux application, because they can help us avoid unnecessary re-renders, and also avoid doing potentially complex or expensive calculations if the input data hasn't changed.

Investigating the Posts List

If we go back to our `<PostsList>` and try clicking a reaction button on one of the posts while capturing a React profiler trace, we'll see that not only did the `<PostsList>` and the updated `<PostExcerpt>` instance render, all of the `<PostExcerpt>` components rendered:



Why is that? None of the other posts changed, so why would they need to re-render?

React's default behavior is that when a parent component renders, React will recursively render all child components inside of it! The immutable update of one `post` object also created a new `posts` array. Our `<PostsList>` had to re-render because the `posts` array was a new reference, so after it rendered, React continued downwards and re-rendered all of the `<PostExcerpt>` components too.

This isn't a serious problem for our small example app, but in a larger real-world app, we might have some very long lists or very large component trees, and having all those extra components re-render might slow things down.

There are a few different ways we could optimize this behavior in `<PostsList>`.

First, we could wrap the `<PostExcerpt>` component in `React.memo()`, which will ensure that the component inside of it only re-renders if the props have actually changed. This will actually work quite well - try it out and see what happens:

`features/posts/PostsList.js`

```
JavaScript
let PostExcerpt = ({ post }) => {
  // omitted logic
}

PostExcerpt = React.memo(PostExcerpt)
```

Another option is to rewrite `<PostsList>` so that it only selects a list of post IDs from the store instead of the entire `posts` array, and rewrite `<PostExcerpt>` so that it receives a `postId` prop and calls `useSelector` to read the post object it needs. If `<PostsList>` gets the same list of IDs as before, it won't need to re-render, and so only our one changed `<PostExcerpt>` component should have to render.

Unfortunately, this gets tricky because we also need to have all our posts sorted by date and rendered in the right order. We could update our `postsSlice` to keep the array sorted at all times, so we don't have to sort it in the component, and use a memoized selector to extract just the list of post IDs. We could also customize the comparison function that `useSelector` runs to check the results, like `useSelector(selectPostIds, shallowEqual)`, so that it will skip re-rendering if the contents of the IDs array haven't changed.

The last option is to find some way to have our reducer keep a separate array of IDs for all the posts, and only modify that array when posts are added or removed, and do the same rewrite of `<PostsList>` and `<PostExcerpt>`. This way, `<PostsList>` only needs to re-render when that IDs array changes.

Conveniently, Redux Toolkit has a `createEntityAdapter` function that will help us do just that.

Normalizing Data

You've seen that a lot of our logic has been looking up items by their ID field. Since we've been storing our data in arrays, that means we have to loop over all the items in the array using `array.find()` until we find the item with the ID we're looking for.

Realistically, this doesn't take very long, but if we had arrays with hundreds or thousands of items inside, looking through the entire array to find one item becomes wasted effort. What we need is a way to look up a single item based on its ID, directly, without having to check all the other items. This process is known as "normalization".

Normalized State Structure

"Normalized state" means that:

- We only have one copy of each particular piece of data in our state, so there's no duplication.
- Data that has been normalized is kept in a lookup table, where the item IDs are the keys, and the items themselves are the values.
- There may also be an array of all of the IDs for a particular item type.

JavaScript objects can be used as lookup tables, similar to "maps" or "dictionaries" in other languages. Here's what the normalized state for a group of `user` objects might look like:

```
JavaScript
{
  users: {
    ids: ["user1", "user2", "user3"],
    entities: {
      "user1": {id: "user1", firstName, lastName},
      "user2": {id: "user2", firstName, lastName},
      "user3": {id: "user3", firstName, lastName},
    }
  }
}
```

This makes it easy to find a particular `user` object by its ID, without having to loop through all the other user objects in an array:

```
JavaScript
const userId = 'user2'
const userObject = state.users.entities[userId]
```

Managing Normalized State with `createEntityAdapter`

Redux Toolkit's `createEntityAdapter` API provides a standardized way to store your data in a slice by taking a collection of items and putting them into the shape of `{ ids: [], entities: {} }`. Along with this predefined state shape, it generates a set of reducer functions and selectors that know how to work with that data.

This has several benefits:

- We don't have to write the code to manage the normalization ourselves
- `createEntityAdapter`'s pre-built reducer functions handle common cases like "add all these items", "update one item", or "remove multiple items"
- `createEntityAdapter` can keep the ID array in a sorted order based on the contents of the items, and will only update that array if items are added / removed or the sorting order changes.

`createEntityAdapter` accepts an options object that may include a `sortComparer` function, which will be used to keep the item IDs array in sorted order by comparing two items (and works the same way as `Array.sort()`).

It returns an object that contains a set of generated reducer functions for adding, updating, and removing items from an entity state object. These reducer functions can either be used as a case reducer for a specific action type, or as a "mutating" utility function within another reducer in `createSlice`.

The adapter object also has a `getSelectors` function. You can pass in a selector that returns this particular slice of state from the Redux root state, and it will generate selectors like `selectAll` and `selectById`.

Finally, the adapter object has a `getInitialState` function that generates an empty `{ids: [], entities: {}}` object. You can pass in more fields to `getInitialState`, and those will be merged in.

Updating the Posts Slice

With that in mind, let's update our `postsSlice` to use `createEntityAdapter`:

`features/posts/postsSlice.js`

```
JavaScript
import {
  createEntityAdapter
  // omitted other imports
} from '@reduxjs/toolkit'

const postsAdapter = createEntityAdapter({
  sortComparer: (a, b) => b.date.localeCompare(a.date)
})

const initialState = postsAdapter.getInitialState({
  status: 'idle',
  error: null
})

// omitted thunks

const postsSlice = createSlice({
  name: 'posts',
  initialState,
  reducers: {
    reactionAdded(state, action) {
```

```
      const { postId, reaction } = action.payload
      const existingPost = state.entities[postId]
      if (existingPost) {
        existingPost.reactions[reaction]++
      }
    },
    postUpdated(state, action) {
      const { id, title, content } = action.payload
      const existingPost = state.entities[id]
      if (existingPost) {
        existingPost.title = title
        existingPost.content = content
      }
    }
  },
  extraReducers(builder) {
    // omitted other reducers

    builder
      .addCase(fetchPosts.fulfilled, (state, action) => {
        state.status = 'succeeded'
        // Add any fetched posts to the array
        // Use the `upsertMany` reducer as a mutating update
utility
        postsAdapter.upsertMany(state, action.payload)
      })
      // Use the `addOne` reducer for the fulfilled case
      .addCase(addNewPost.fulfilled, postsAdapter.addOne)
  }
})

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

export default postsSlice.reducer

// Export the customized selectors for this adapter using
`getSelectors`
export const {
```

```
  selectAll: selectAllPosts,
  selectById: selectPostById,
  selectIds: selectPostIds
  // Pass in a selector that returns the posts slice of state
} = postsAdapter.getSelectors(state => state.posts)

export const selectPostsByUser = createSelector(
  [selectAllPosts, (state, userId) => userId],
  (posts, userId) => posts.filter(post => post.user === userId)
)
```

There's a lot going on there! Let's break it down.

First, we import `createEntityAdapter`, and call it to create our `postsAdapter` object. We know that we want to keep an array of all post IDs sorted with the newest post first, so we pass in a `sortComparer` function that will sort newer items to the front based on the `post.date` field.

`getInitialState()` returns an empty `{ids: [], entities: {}}` normalized state object. Our `postsSlice` needs to keep the `status` and `error` fields for loading state too, so we pass those in to `getInitialState()`.

Now that our posts are being kept as a lookup table in `state.entities`, we can change our `reactionAdded` and `postUpdated` reducers to directly look up the right posts by their IDs, instead of having to loop over the old `posts` array.

When we receive the `fetchPosts.fulfilled` action, we can use the `postsAdapter.upsertMany` function to add all of the incoming posts to the state, by passing in the draft `state` and the array of posts in `action.payload`. If there's any items in `action.payload` that already existing in our state, the `upsertMany` function will merge them together based on matching IDs.

When we receive the `addNewPost.fulfilled` action, we know we need to add that one new post object to our state. We can use the adapter functions as reducers directly, so we'll pass `postsAdapter.addOne` as the reducer function to handle that action.

Finally, we can replace the old hand-written `selectAllPosts` and `selectPostById` selector functions with the ones generated by `postsAdapter.getSelectors`. Since the selectors are called with the root Redux state object, they need to know where to find our posts data in the Redux state, so we pass in a small selector that returns `state.posts`. The generated selector functions are always called `selectAll` and `selectById`, so we can use ES6 destructuring

syntax to rename them as we export them and match the old selector names. We'll also export `selectPostIds` the same way, since we want to read the list of sorted post IDs in our `<PostsList>` component.

Optimizing the Posts List

Now that our posts slice is using `createEntityAdapter`, we can update `<PostsList>` to optimize its rendering behavior.

We'll update `<PostsList>` to read just the sorted array of post IDs, and pass `postId` to each `<PostExcerpt>`:

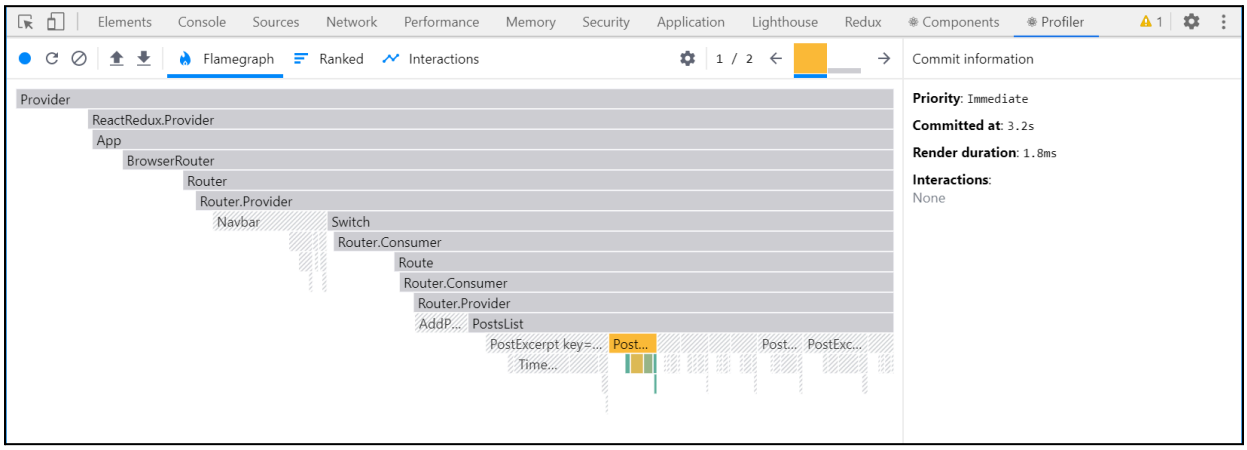
`features/posts/PostsList.js`

JavaScript

```
// omitted other imports

import {
  selectAllPosts,
  fetchPosts,
  selectPostIds,
  selectPostById
} from './postsSlice'

let PostExcerpt = ({ postId }) => {
  const post = useSelector(state => selectPostById(state,
postId))
  // omitted rendering logic
}

export const PostsList = () => {
  const dispatch = useDispatch()
  const orderedPostIds = useSelector(selectPostIds)

  // omitted other selections and effects

  if (postStatus === 'loading') {
    content = <Spinner text="Loading..." />
```

```
  } else if (postStatus === 'succeeded') {
    content = orderedPostIds.map(postId => (
      <PostExcerpt key={postId} postId={postId} />
    ))
  } else if (postStatus === 'error') {
    content = <div>{error}</div>
  }

  // omitted other rendering
}
```

Now, if we try clicking a reaction button on one of the posts while capturing a React component performance profile, we should see that *only* that one component re-rendered:



Converting Other Slices

We're almost done. As a final cleanup step, we'll update our other two slices to use `createEntityAdapter` as well.

Converting the Users Slice

The `usersSlice` is fairly small, so we've only got a few things to change:

`features/users/usersSlice.js`

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

const usersAdapter = createEntityAdapter()

const initialState = usersAdapter.getInitialState()

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

const usersSlice = createSlice({
  name: 'users',
  initialState,
  reducers: {},
  extraReducers(builder) {
    builder.addCase(fetchUsers.fulfilled, usersAdapter.setAll)
  }
})

export default usersSlice.reducer

export const { selectAll: selectAllUsers, selectById:
selectUserById } =
  usersAdapter.getSelectors(state => state.users)
```

The only action we're handling here always replaces the entire list of users with the array we fetched from the server. We can use `usersAdapter.setAll` to implement that instead.

Our `<AddPostForm>` is still trying to read `state.users` as an array, as is `<PostAuthor>`. Update them to use `selectAllUsers` and `selectUserById`, respectively.

Converting the Notifications Slice

Last but not least, we'll update `notificationsSlice` as well:

`features/notifications/notificationsSlice.js`

JavaScript

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

import { client } from '../../api/client'

const notificationsAdapter = createEntityAdapter({
  sortComparer: (a, b) => b.date.localeCompare(a.date)
})

// omitted fetchNotifications thunk

const notificationsSlice = createSlice({
  name: 'notifications',
  initialState: notificationsAdapter.getInitialState(),
  reducers: {
    allNotificationsRead(state, action) {
      Object.values(state.entities).forEach(notification => {
        notification.read = true
      })
    }
  },
  extraReducers(builder) {
    builder.addCase(fetchNotifications.fulfilled, (state, action)
=> {
      notificationsAdapter.upsertMany(state, action.payload)
      Object.values(state.entities).forEach(notification => {
        // Any notifications we've read are no longer new
        notification.isNew = !notification.read
      })
    })
  }
})
```

```
export const { allNotificationsRead } =
notificationsSlice.actions

export default notificationsSlice.reducer

export const { selectAll: selectAllNotifications } =
  notificationsAdapter.getSelectors(state => state.notifications)
```

We again import `createEntityAdapter`, call it, and call `notificationsAdapter.getInitialState()` to help set up the slice.

Ironically, we do have a couple places in here where we need to loop over all notification objects and update them. Since those are no longer being kept in an array, we have to use `Object.values(state.entities)` to get an array of those notifications and loop over that. On the other hand, we can replace the previous fetch update logic with `notificationsAdapter.upsertMany`.

And with that... we're done learning the core concepts and functionality of Redux Toolkit!

As we are sure you've realized by now, Redux Toolkit is an incredibly powerful package that can help you handle many of the common tasks you'll face as a developer. It has many tools that you will need to familiarize yourself with over the course of time to be a true Redux master.

Part Four Recap

The final Redux recap! This section included the following key concepts centered around performance:

- **Memoized selector functions can be used to optimize performance.**
  - Redux Toolkit re-exports the `createSelector` function from Reselect, which generates memoized selectors.
  - Memoized selectors will only recalculate the results if the input selectors return new values.
  - Memoization can skip expensive calculations, and ensure the same result references are returned.
- **There are multiple patterns you can use to optimize React component rendering with Redux.**
  - Avoid creating new object/array references inside of `useSelector` - those will cause unnecessary re-renders.

    - Memoized selector functions can be passed to `useSelector` to optimize rendering.
    - `useSelector` can accept an alternate comparison function like `shallowEqual` instead of reference equality.
    - Components can be wrapped in `React.memo()` to only re-render if their props change.
    - List rendering can be optimized by having list parent components read just an array of item IDs, passing the IDs to list item children, and retrieving items by ID in the children.
- **Normalized state structure is a recommended approach for storing items.**
    - "Normalization" means no duplication of data, and keeping items stored in a lookup table by item ID.
    - Normalized state shape usually looks like `{ids: [], entities: {}}`.
- **Redux Toolkit's `createEntityAdapter` API helps manage normalized data in a slice.**
    - Item IDs can be kept in sorted order by passing in a `sortComparer` option.
    - The adapter object includes:
        - `adapter.getInitialState`, which can accept additional state fields like loading state.
        - Prebuilt reducers for common cases, like `setAll`, `addMany`, `upsertOne`, and `removeMany`.
        - `adapter.getSelectors`, which generates selectors like `selectAll` and `selectById`.

Here's what your application should look like: Part Four Completed CodeSandbox.