Redux

Expense Tracker

This project—a budgeting and expense tracking app—allows you to practice refactoring with Redux Toolkit. The app allows you to set budgets for various categories, such as food and transportation, and track transactions in those categories. It then sums your spending in each category to calculate the amount of money that remains to be spent.

To help you to understand how the data of the application works, consider an example of the Redux store’s state:

{  
  budgets: [   
    { category: 'housing', amount: 400 },  
    { category: 'food', amount: 100 },  
    ...  
  ],  
  transactions: {  
    housing: [   
      {   
        category: 'housing',   
        description: 'rent',   
        amount: 400,   
        id: 123   
      }  
    ],  
    food: [   
      {   
        category: 'food',   
        description: 'groceries on 1/12/2021',   
        amount: 50,   
        id: 456   
      },  
      {   
        category: 'food',   
        description: 'dinner on 1/16/2021',   
        amount: 12,   
        id: 789   
      },  
    ]  
  }  
   
   
}

You will work primarily in **budgetsSlice.js** and **transactionsSlice.js** where reducers and action creators are currently programmed by hand. Your task will be to refactor this project using a slice-based approach to produce the app’s actions and reducers.

Before you get started, spend some time using the app in its current implementation to ensure you understand how it’s supposed to work. Set a budget of $300 for food, create a $20 food transaction, and then check the food budget again to see how much you have left to spend. As you progress through the project, take note of the ways that Redux Toolkit simplifies your code.

**Create a Budgets Slice**

1.

Without Redux Toolkit, you have to write all your action creators and reducers by hand. Redux Toolkit’s [createSlice()](https://redux-toolkit.js.org/api/createSlice) function generates action creators and reducers for you based on the inputs you give it.

Not only does createSlice() reduce the amount of code you have to write by automatically generating action creators and reducers, it also simplifies your reducers by allowing you to write mutating logic inside your reducers.

At the top of **budgetsSlice.js**:

1. Import createSlice from @reduxjs/toolkit.

Import createSlice() from Redux Toolkit using the following syntax.

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

2.

Next, you are going to define a slice by calling createSlice() with a configuration object containing the required name, initialState, and reducers properties.

1. Define a variable, budgetsSlice, and initialize it with a call to createSlice(), passing in an empty configuration object. Do this right after the line defining initialState.
2. Slices are conventionally named for the resource whose state they manage. This slice manages budgets and should be named accordingly. To give the slice a name, add a name property to the configuration object and set it equal to 'budgets'.
3. Add an initialState property to the configuration object, and set it equal to the variable initialState that we’ve defined for you.
4. Lastly, you’ll need to include a reducers property in the configurations object. For now, set it equal to an empty object.

Your call to createSlice() should look something like this:

const budgetsSlice = createSlice({  
  name: // slice name here,  
  initialState: // initial state here,  
  reducers: {}   
})

3.

In **budgetsSlice.js**, which we originally wrote without Redux Toolkit, you’ll see an editBudget() action creator. Currently, the action dispatched by that action creator will be processed in the 'budgets/editBudget' case of the budgetsReducer() we’ve provided. Open **components/Budget.js** where you can see this action being dispatched like so:

dispatch(editBudget({category: budget.category, amount: amount}))

createSlice() automatically generates action creators and action types based on the case reducer functions we include in the reducers property. Once we define an editBudget case reducer, we will be able to delete our standalone action creators and reducers, and greatly simplify our code in the process.

1. Add an editBudget property to the reducers object passed to createSlice().
2. Set editBudget equal to a case reducer that receives two arguments—state and action . action.payload will have a category and amount property.
3. editBudget should update the state by finding the budget object whose .category value matches action.payload.category and changing with the .amount value to action.payload.amount.

The reducers property you pass to createSlice() should look something like:

reducers: {  
  editBudget: (state, action) => {  
    // find the budget object in state whose category matches the `category` property on `action.payload` and update the amount of the budget object to match the `amount` on `action.payload`. Remember, `createSlice()` uses `Immer` internally, so you are free to write mutating logic that updates the slice's state.  
  }  
}

4.

Now that you’ve implemented budgetsSlice, you’ll want to delete your old code and clean up your exports.

1. Delete the stand-alone editBudget. At the bottom of the file **budgetsSlice.js**, export the editBudget action creator generated by createSlice() and stored in budgetsSlice.
2. Delete the stand-alone budgetsReducer, and update the export default statement to export the reducer generated by createSlice() and stored in budgetsSlice.

Once you’ve completed this task, you should be able to edit budgets and see your changes reflected in the app.

The editBudget action creator generated by createSlice() is accessible as a property on budgetsSlice.actions, and the budgets reducer is accessible via the budgetsSlice.reducer property. Your code to export these values should look something like this:

export const { myActionCreator } = mySlice.actions;  
export default mySlice.reducer;

**Create a Transactions Slice**

5.

Great work! Now that you’ve refactored the budgets slice with Redux Toolkit, you should do the same to the transactions slice, which is responsible for storing all of the user’s transactions organized by category. The slice handles adding new transactions and deleting existing transactions, and its state is an object structured like this:

transactions = {  
  housing: [   
    {   
      category: 'housing',   
      description: 'rent',   
      amount: 400,   
      id: 123   
    }  
  ],  
  food: [   
    {   
      category: 'food',   
      description: 'groceries on 1/12/2021',   
      amount: 50,   
      id: 456   
    },  
    {   
      category: 'food',   
      description: 'dinner on 1/16/2021',   
      amount: 12,   
      id: 789   
    },  
  ]  
}

In **transactionsSlice.js**:

1. Import createSlice from @reduxjs/toolkit.

Import createSlice() from Redux Toolkit using the following syntax.

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

6.

Next, you are going to define a slice by calling createSlice() with a configuration object containing the required name, initialState, and reducers properties.

1. Define a variable, transactionsSlice, and initialize it with a call to createSlice(), passing in an empty configuration object.
2. Add a name property to the configuration object and set it equal to 'transactions'.
3. Add an initialState property to the configuration object, and set it equal to the variable initialState that we’ve defined for you.
4. Lastly, you’ll need to include a reducers property in the configurations object. For now, set it equal to an empty object.

Your call to createSlice() should look something like this:

const transactionsSlice = createSlice({  
  name: // slice name here,  
  initialState: // initial state here,  
  reducers: {}   
})

7.

Since we originally developed this project without Redux Toolkit, you’ll see two stand-alone action creators: addTransaction() and deleteTransaction(). Each of these action creators will receive an action.payload value that is a transaction object like so:

transaction = {  
  category: 'housing',   
  description: 'rent for January',   
  amount: 400,   
  id: 123  
}

Open **components/TransactionForm.js** and **components/Transaction.js** where you can see how addTransaction() and deleteTransaction() are dispatched, respectively. Currently, the actions dispatched by these action creators will be processed by the transactionsReducer we’ve provided.

Your task is to replace these stand-alone action creators and the reducer with case reducers defined in the object passed to createSlice().

1. Add an addTransaction property to the reducers object passed to createSlice().
2. Set addTransaction equal to a case reducer that receives two arguments—state and action. It should add the new transaction object (action.payload) to the correct category’s transaction list in the transactions state object.
3. Add a deleteTransaction property to the reducers object passed to createSlice().
4. Set deleteTransaction equal to a case reducer that receives two arguments—state and action. It should delete the old transaction (action.payload) from the correct category’s transaction list in the transactions state object.

The reducers property you pass to createSlice() should look something like:

reducers: {  
  addTransaction: (state, payload) => {  
    // 1. Find the category in `state` that matches the `category` property on `action.payload`   
   
    // 2. Add the new transaction (`action.payload`) to that category's transaction array.  
  },  
  deleteTransaction: (state, payload) => {  
    // 1. Find the category in `state` that matches the `category` property on `action.payload`

  // 2.  Filter out the old transaction (the transaction with an `id` matching the `id` property on `action.payload`) from that category's transaction array.  
  }  
}

8.

Now that you’ve implemented transactionsSlice, you’ll want to delete your old code and clean up your exports.

1. Delete the stand-alone addTransaction and deleteTransaction, and export the addTransaction and deleteTransaction action creators generated by createSlice()and stored in transactionsSlice.
2. Delete the stand-alone transactionsReducer, and update the export default statement to export the reducer generated by createSlice() and stored in transactionsSlice.

At this point, you should be able to add and delete transactions and see your changes reflected in the transactions list as well as in the “Funds Remaining” field for each budget.

The addTransaction and deleteTransaction action creators generated by createSlice() are accessible as properties on transactionsSlice.actions, and the transactions reducer is accessible via the transactionsSlice.reducer property. Your code to export these values should look something like this:

export const { anActionCreator, anotherActionCreator } = mySlice.actions;  
export default mySlice.reducer;