Redux Notes

Mohamed Emary

September 14, 2024

1 Redux Vs Redux Toolkit

Redux	Redux Toolkit
Complex (Manual Configuration)	Simple (Built-in Functions)
Hard to organize (Apply separation of concerns)	Easy to organize (Slices)
Need to be done manually	Automatically caches the data

2 Redux

In redux you can create only one store but divide that store into multiple slices. Each slice can have its own reducer, action, and selector.

3 Important Notes

Redux has **store** that you should provide to all your application in `App.jsx` via `<Provider>` component and it takes `store` as a prop.

```
<Provider store={reduxStore}></Provider>
```

To create that store you create a component `reduxStore.js` and create a store using `configureStore` function.

```
const reduxStore = configureStore({
  reducer:{
    slice1: slice1Reducer,
    slice2: slice2Reducer,
    ...
  },
});

export default reduxStore;
```

But first we need to create reducers for each slice.

> Note 1: The slices will be called by `slice1` and `slice2` not by `slice1Reducer` and `slice2Reducer`.
>
> Note 2: You can use just one value and both the key and the value will be the same. for example `reducer: {slice1Reducer, slice2Reducer}`

Lets create a counter slice.

```
const counterSlice = createSlice({
  name: "counter", // unique name

  initialState: {
    name: "Mohamed",
    counter: 10,
  },

  reducers: {
    increment: (state, action) => {
      state.counter += action.payload;
    },

    decrement: (state) => {
      state.counter--;
    },
  },
});

export const { increment, decrement } = counterSlice.actions;

export default counterSlice.reducer; // This is what we will use in the
 ↪  store.
```

The name must be unique because it will be used to differentiate between slices.

In the code above we have two similar words `reducers` and `.reducer` the first one is an object that contains all the reducers (functions that can change the state) for this slice and the second one is the reducer itself that we will use in the store.

The `reducers` is an object (key, value) the key is the name of the function and the value is the function itself. The function accepts two arguments the first one is the `state` which is an object that contains all the data in the slice (`name` and `counter`) and the second one is the `action` which is an object that contains the `type` and the `payload`.

When we pass a value to the `increment` function it will be passed to the `action` object and we can access it using `action.payload`. That value can be anything (string, number, object, array, ...).

The slice you see above handles synchronous actions. What if we want to handle asynchronous actions?

This is another slice that handles asynchronous actions.

```
export const getPizza = createAsyncThunk("apiSlice/getPizza", () => {
```

```
  return fetch("https://ecommerce.routemisr.com/api/v1/brands").then((res)
  ↪  =>
    res.json()
  );
});

export default createSlice({
  name: "apiSlice", // unique name
  initialState: {
    data: [],
    isLoading: false,
    isError: false,
  },

  extraReducers: (builder) => {
    builder.addCase(getPizza.fulfilled, (state, action) => {
      console.log("called fulfilled");
      state.data = action.payload;
      state.isError = false;
      state.isLoading = false;
      console.log(action.payload);
      console.log(state);
    });

    builder.addCase(getPizza.pending, (state, action) => {
      console.log("called pending");
      state.isLoading = true;
      console.log(action.payload);
      console.log(state);
    });

    builder.addCase(getPizza.rejected, (state, action) => {
      console.log("called rejected");
      state.isError = true;
      state.isLoading = false;
      console.log(action.payload);
      console.log(state);
    });
  },
}).reducer;
```

First we need to create an asynchronous action using `createAsyncThunk` function. The first argument is the name of the action and the second argument is the function that will be called when the action is dispatched.

> Note 3: To connect that action with the slice the name of the action must consist of the slice name (the unique name) and the action name separated by a `/`, for example `apiSlice/getPizza`.

The `extraReducers` takes a function that takes a `builder` object. You can use that builder object to add cases for the action, for example since our action name is `getPizza` we can add three cases `getPizza.fulfilled`, `getPizza.pending`, and `getPizza.rejected`, and do

different things accordingly.

`getPizza.fulfilled` handles the success case, `getPizza.pending` handles the loading case, and `getPizza.rejected` handles the error case.

Now lets see what our store looks like after adding the two slices.

```
const reduxStore = configureStore({
  reducer: {
    counter: counterReducer,
    api: apiReducer,
  },
});

export default reduxStore;
```

Now we have two slices `counter` and `api` in our store that we can use in a component.

`Home.jsx` component:

```
export default function Home() {
  const { counter, name } = useSelector((store) => store.counter);
  const dispatch = useDispatch();

  const res = useSelector((store) => store.api);

  useEffect(() => {
    dispatch(getPizza());
  }, []);

  return (
    <>
      <div className="container">
        {console.log("res = ", res)}
        <h1 className="font-semibold text-2xl text-center">{name}</h1>
        <h1 className="font-semibold text-2xl text-center">{counter}</h1>

        <button
          className="px-3 py-2 bg-slate-500 m-2 rounded-lg"
          onClick={() => dispatch(increment(5))}>
          +
        </button>
        <button
          className="px-3 py-2 bg-slate-500 m-2 rounded-lg"
          onClick={() => dispatch(decrement())}>
          -
        </button>
      </div>
    </>
  );
}
```

Notice that to be able to use any of the `reducers` functions we need to **call** it inside a `dispatch` function.

To access different data stores we use `useSelector` function and pass the `store` object to it and then we can access the data from it.