Technische Universität München
Fakultät für Informatik

Exercise Sheet 5: Testability and React Redux

Welcome to the fifth week of Advanced Topics of Software Engineering. This week we will revisit *testability*, a software quality attribute concerned with characteristics that affect the test effort necessary to test a piece of software. A low degree of testability is thus undesirable as it comes with high costs for software testing.

In the theoretical exercise, we will practice identifying factors that influence testability, analyzing methods to assess testability and examining problems in test selection criteria.

In this week's practical exercises, we are going to learn how to use React Redux for state management and how to consume a REST API.

Theoretical Exercise

Exercise 1: Testability

In the lecture you learned about the (rather scientific) concept of testability, different factors determining testability of programs, and how to assess testability. As you know, testability concerns characteristics of the software under consideration that affect the effort needed to test it.

1. Factors Influencing Testability
    - Name and explain the two main factors affecting testability introduced in the lecture. What can be a common problem when trying to improve testability based on these in practice?
    - Reason why two of the already introduced principles (or guidelines) for modular software design could increase testability of a program.
    - What could cause a system to behave in a non-deterministic way during testing (i.e. *flaky tests*)?
2. Approach for assessing testability
    - The lecture introduced the concept of information loss and the Domain/range ratio (DRR). Should you be especially careful when testing functions with low or high DRR and why?
3. Test selection criteria
    - Recall a criterion/metric from the lecture that favors one test suite over another. What is a problem of this criterion with respect to the ultimate goal of delivering failure-free functioning of the code?

Practical Exercise

In React, one of the major problems is state management. States can store anything from API responses to UI data and with every newly added component, the management of this ever-changing state becomes harder. With a chain of components that update each other, you can lose control of your app regarding why, when, or how its state is updated. As a result, you no longer understand what happens in your app.

Redux is a tool that helps you to deal with applications that have complex, shared state management tasks. In short, Redux is a *predicatable* state container. The state of your application is saved in a store with Redux, and each component can access any state it requires from this store. Redux becomes most useful when:

- same app state is utilized in many different components,
- frequent updates to complex app state update logic are required,
- you need to keep track of how app state is being updated over time,
- codebase is worked on by many people.

However, you may not be needing Redux and it may be better not to use it as it is not designed to be the shortest or fastest way to write code. You can read https://medium.com/@dan_abramov/you-might-not-need-redux-be46360cf367 to get a better understanding of when to and when not to use Redux.

How Redux Works?
In Redux, a global *store* holds the application state and each component can access the stored state without sending props down to another component. Besides store, there are two more building blocks of Redux: *actions* and *reducers*.

Actions are events that enable your application to interact with your Redux store. They are useful for passing data from user interactions, API calls, or form submissions to your store and eventually to your states. To send an action, the *dispatch()* method of your store variable must be used. Actions are simple JavaScript objects that *must* have **type** and **payload** fields. While the former field indicates the type of action to be carried out, the latter contains the data.

Reducers, on the other hand, are like event handlers that deal with events based on the received action type. They receive the current state and an action object and after carrying out the necessary updates on the state, return the new state. An important remark is that reducers do not modify the existing state, but instead, they do **immutable updates** by copying the existing state and making modifications to the copied values.

Visit https://redux.js.org/tutorials/fundamentals/part-2-concepts-data-flow to get a deeper understanding of the background concepts of Redux.

What is Redux Toolkit?
Redux Toolkit is a package that aims to standardize the way Redux logic is written and help make your Redux code better. The toolkit includes several APIs that reduce the amount of boilerplate code required by Redux and makes it less complex to configure a Redux store. Visit https://redux-toolkit.js.org/introduction/getting-started to read more about the Redux Toolkit and the available APIs.

Exercise 2: Introduction to React Redux

1. Run the following command to start a new React + Redux app that takes advantage of Redux Toolkit.
   `npx create-react-app <your_app_name> -template redux`
   When you run the app with `npm start`, you will see that a counter application is already there by default.
2. Identify the component(s) (other than App.js) of your application and try to understand which data is stored in the local state and which in the global state. How does the component access the data stored in the global state?

3. Open the respective slice file for the component you have identified. First, take a look at the call to the *createSlice*[1] API. This API is responsible for accepting an object of reducers, a slice name, and an initial state, and automatically generating actions that correspond to the reducers. Without slices, you would have to separately define your actions.

4. In counterSlice, you will see that there are 3 reducers (increment, decrement, incrementByAmount) defined. Describe what each does to the state. Identify where their respective actions are dispatched.

5. Finally, check the *store.js* file of your application. This is where the reducers you defined are combined to create your global Redux store. For every slice you have, you need to add a reducer to this store configuration.

   Note: *As you may have noticed, there are some functions (thunks) in counterSlice that we have skipped for now. We will go back to them in the exercise about consuming an API with React Redux. To get a better understanding of the counter application, visit https://redux.js.org/tutorials/essentials/part-2-app-structure.*

Exercise 3: Adding Redux Logic to the Quoter Application

In this exercise, we will add React Redux as a dependency to our Quoter application from the previous week's exercise. If you are having problems with your application's design, feel free to use the following code piece. (Don't forget to do the necessary MUI component imports.)

```
<Container component="main" maxWidth="sm" sx={{ mb: 4 }}>
  <Paper
    variant="outlined"
    sx={{ my: { xs: 3, md: 6 }, p: { xs: 2, md: 3 } }}
  >
    <Typography component="h1" variant="h4" align="center" marginBottom={5}>
      Welcome to Quoter
    </Typography>
    <Typography component="h1" variant="h5" style={{ fontStyle: "italic" }} marginBottom={5}>
      "{currentQuote}"
    </Typography>
    <div style={{display:"flex", marginBottom:20}}>
      <TextField fullWidth id="quote" size="small" label="New Quote" name="quote" onChange={
          handleChange} value={newQuote}/>
      <Button variant="contained" size="small" style={{marginLeft:10}} onClick={handleAdd}>Add
          </Button>
    </div>
    <Button variant="contained" size="small" color="secondary" fullWidth onClick={
        handleRandomize}>Show Me Another Quote</Button>
  </Paper>
</Container>
```

1. Open your application's directory and run the following commands:
   ```
   npm install react-redux
   npm install @reduxjs/toolkit
   ```

2. You have to first configure your Redux store. Take a look at the index.js file of the counter application and add the missing pieces to the index.js file of your Quoter application.
   Hint: *Wrap your App component with a* Provider *which you pass the store variable you imported from your actual store implementation (store.js).*

3. Create a "views" folder and inside this folder create a folder for your Quote component and slice (quoteSlice.js).
   Tip: *Don't forget to update the import statement in App.js, after changing the location of your Quote component.*

4. In your Quote component, you should ideally have 3 local state variables: *newQuote*, *currentQuote*, *quotes*. Based on what you have seen in the counter application, determine which variables should be stored in your global Redux store.

5. Now, implement your quoteSlice. Ideally, it should have at least 2 reducers. One for adding a quote to the quotes list and the other for setting the current quote.
   Tip: *Don't forget to configure your store once you are done with your slice.*

[1] https://redux-toolkit.js.org/api/createSlice

6. Finally, update your component. It should use the data from the global state (where it is necessary) and should dispatch actions for handling events that update the global state.
   Note: *Once all necessary updates are made, the application should work just as it was working without the Redux logic. Although the Quoter application may not be the best use case of Redux, it should give you an idea about how practical it can be to use Redux, when state management becomes more complex.*

Exercise 4: Consuming an API with React Redux

So far we have only worked with data that are directly included in our React Redux application. However, this is not the case in most apps as they need to work with data from a server by making HTTP API calls. To write async logic in React Redux, we are going to use *redux-thunk* middleware. The configureStore API of Redux Toolkit automatically sets up redux-thunk middleware, thus we can start using it right away in our slices.

1. Open counterSlice.js file from your counter application. You will see that *incrementAsync* function uses the "createAsyncThunk" API. This API accepts two arguments: a type prefix (in this case "counter/fetchCount") and a callback function that handles the HTTP API call and returns a promise. Describe what this thunk does.
   Now take a look at the extraReducers field in counterSlice.js. This field lets you handle actions that are either generated by thunks or defined elsewhere. You'll see that actions dispatched by incrementAsync thunk ("pending", "fulfilled") are listened here. These actions are automatically attached to your thunk since it is created with createAsyncThunk API. Visit https://redux.js.org/tutorials/essentials/part-5-async-logic to get a better understanding of how async logic can be implemented in React Redux.
2. You can also write thunks without using the createAsyncThunk API. A thunk expects two arguments: *dispatch* and *getState* method of the Redux store. Take a look at the *incrementIfOdd* function in counterSlice.js. Describe what it does.
3. Now, let's go back to our Quoter application. We want to add new functionality which shows a random quote fetched from the Quotable[2] API. Choose one of the available endpoints (e.g. https://api.quotable.io/random).
4. Add a new button to your Quoter application that dispatches an async action for fetching a random quote from the Quotable API. The fetched quote should be displayed on the view right away. The previous functionality of the application should also still work (e.g. the user should be able to save the quote by adding it as a new quote). You can either use the createAsyncThunk API or write your own thunk.

[2] https://github.com/lukePeavey/quotable