12 Redux and Hooks

Exercise 1: Handling state with Redux

State management with Redux is actually really similar to using a Reducer Hook. We first define the state object, then actions, and finally, our reducers. An additional pattern in Redux is to create functions that return action objects, so-called action creators.
Furthermore, we need to wrap our whole app with a `Provider` component, and connect components to the Redux store in order to be able to use Redux state and action creators.

Step 1: Installing Redux

First of all, we have to install Redux, React Redux, and Redux Thunk. Let us look at what each one does individually:

- Redux itself just deals with JavaScript objects, so it provides the store, deals with actions and action creators, and handles reducers.
- React Redux provides connectors in order to connect Redux to our React components.
- Redux Thunk is a middleware that allows us to deal with asynchronous requests in Redux.

Using **Redux** in combination with **React** offloads global state management to **Redux**, while **React** deals with rendering the application and local state:

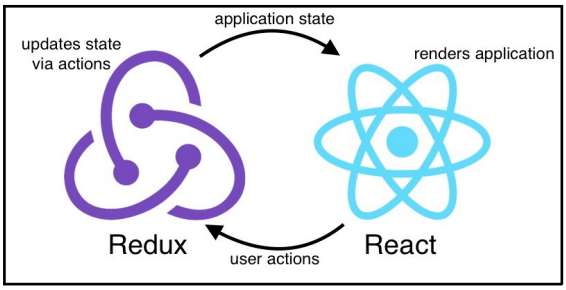

Illustration of how React and Redux work together

To install Redux and React Redux, we are going to use `npm`. Execute the following command:

```
> npm install --save redux react-redux redux-thunk
```

Now that all of the required libraries are installed, we can start setting up our Redux store.

Step 2: Defining state, actions, and reducers

The first step in developing a Redux application is defining the state, then the actions that are going to change the state, and finally, the reducer functions, which carry out the state modification. In our ToDo application, we have already defined the state, the actions, and the reducers, in order to use the Reducer Hook. Here, we simply recap what we defined in the previous chapter.

State

The full state object of our ToDo app consists of two keys: an array of todo items, and a string, which specifies the currently selected `filter` value. The initial state looks as follows:

```
{
    "todos": [
        { "id": 1, "title": "Write React Hooks book", "completed": true },
        { "id": 2, "title": "Promote book", "completed": false }
    ],
    "filter": "all"
}
```

As we can see, in Redux, the state object contains all of the state that is important to our app. In this case, the application state consists of an array of `todos` and a `filter`.

Actions

Our app accepts the following five actions:

`FETCH_TODOS`: To fetch a new list of todo items—`{ type: 'FETCH_TODOS', todos: [] }`

`ADD_TODO`: To insert a new todo item—`{ type: 'ADD_TODO', title: 'Test ToDo app' }`

`TOGGLE_TODO`: To toggle the `completed` value of a todo item—`{ type: 'TOGGLE_TODO', id: 'xxx' }`

`REMOVE_TODO`: To remove a todo item—`{ type: 'REMOVE_TODO', id: 'xxx' }`

`FILTER_TODOS`: To filter todo items—`{ type: 'FILTER_TODOS', filter: 'completed' }`

Reducers

We defined three reducers—one for each part of our state—and an app reducer to combine the other two reducers. The filter reducer waits for a `FILTER_TODOS` action, and then sets the new filter accordingly. The todos reducer listens to the other todo-related actions, and adjusts the todos array by adding, removing, or modifying elements. The app reducer then combines both reducers, and passes actions down to them. After defining all the elements that are needed to create a Redux application, we can now set up the Redux store.

Step 3: Setting up the Redux store

In order to keep things simple initially, and to show how Redux works, we are not going to use connectors for now. We are simply going to replace the `state` object, and the `dispatch` function that was previously provided by a Reducer Hook, with Redux.

Let's set up the Redux store now:

1. Edit `src/App.js`, and import the `useState` Hook, as well as the `createStore` function from the Redux library:

```
import React, { useState, useEffect, useMemo } from 'react'
import { createStore } from 'redux'
```

2. Below the import statements and before the `App` function definition, we are going to initialize the Redux store. We start by defining the initial state:

```
const initialState = { todos: [], filter: 'all' }
```

3. Next, we are going to use the `createStore` function in order to define the Redux store, by using the existing `appReducer` function and passing the `initialState` object:

```
const store = createStore(appReducer, initialState)
```

Please note that in Redux, it is not best practice to initialize the state by passing it to `createStore`. However, with a Reducer Hook, we need to do it this way. In Redux, we usually initialize state by setting default values in the reducer functions. We are going to learn more about initializing state via Redux reducers later in this chapter.

4. Now, we can get the `dispatch` function from the store:

```
const { dispatch } = store
```

5. The next step is removing the following Reducer Hook definition within the `App` function:

```
    const [ state, dispatch ] = useReducer(appReducer, { todos: [],
filter: 'all' })
```

It is replaced with a simple State Hook, which is going to store our Redux state:

```
    const [ state, setState ] = useState(initialState)
```

6. Finally, we define an Effect Hook, in order to keep the State Hook in sync with the Redux store state:

```
    useEffect(() => {
        const unsubscribe = store.subscribe(() => setState(store.getState()))
        return unsubscribe

    }, [])
```

As we can see, the app still runs in exactly the same way as before. Redux works very similarly to the Reducer Hook, but with more functionality. However, there are slight differences in how actions and reducers should be defined, which we are going to learn about in the following sections.

Step 4: Defining action types

The first step when creating a full Redux application is to define so-called action types. They will be used to create actions in action creators and to handle actions in reducers. The idea here is to avoid making typos when defining, or comparing, the `type` property of actions.

Let's define the action types now:

1. Create a new `src/actionTypes.js` file.
2. Define and export the following constants in the newly created file:

```
export const FETCH_TODOS = 'FETCH_TODOS' export const
ADD_TODO = 'ADD_TODO'
export const TOGGLE_TODO = 'TOGGLE_TODO' export const
REMOVE_TODO = 'REMOVE_TODO' export const FILTER_TODOS =
'FILTER_TODOS'
```

Now that we have defined our action types, we can start using them in action creators and reducers.

Defining action creators

After defining the action types, we need to define the actions themselves. In doing so, we are going to define the functions that will return the action objects. These functions are called action creators, of which there are two types:

- **Synchronous action creators**: These simply return an action object
- **Asynchronous action creators**: These return an `async` function, which will later dispatch an action

We are going to start by defining synchronous action creators, then we are going to learn how to define asynchronous action creators.

Step 5: Defining synchronous action creators

We have already defined the action creator functions earlier, in `src/App.js`. We can now copy them from our `App` component, making sure that we adjust the `type` property in order to use the action type constants, instead of a static string.

Let's define the synchronous action creators now:

1. Create a new `src/actions.js` file.
2. Import all action types, which we are going to need to create our actions:

```
import {
    ADD_TODO, TOGGLE_TODO, REMOVE_TODO, FILTER_TODOS
} from './actionTypes'
```

3. Now, we can define and export our action creator functions:

```
export function addTodo (title) {
    return { type: ADD_TODO, title }
}

export function toggleTodo (id) {
    return { type: TOGGLE_TODO, id }
}

export function removeTodo (id) {
     return { type: REMOVE_TODO, id }
}

export function filterTodos (filter) {
    return { type: FILTER_TODOS, filter }
}
```

As we can see, synchronous action creators simply create and return action objects.

Step 6: Defining asynchronous action creators

The next step is defining an asynchronous action creator for the `fetchTodos` action. Here, we are going to use the `async/await` construct.

We are now going to use an `async` function to define the `fetchTodos` action creator:

1. In `src/actions.js`, first import the `FETCH_TODOS` action type and the `fetchAPITodos` function:

```
import {
    FETCH_TODOS, ADD_TODO, TOGGLE_TODO, REMOVE_TODO, FILTER_TODOS
} from './actionTypes'
import { fetchAPITodos } from './api'
```

2. Then, define a new action creator function, which will return an `async` function that is going to get the `dispatch` function as an argument:

```
export function fetchTodos () {
     return async (dispatch) => {
```

3. In this `async` function, we are now going to call the API function, and `dispatch` our action:

```
         const todos = await fetchAPITodos()
         dispatch({ type: FETCH_TODOS, todos })
    }
}
```

As we can see, asynchronous action creators return a function that will dispatch actions at a later time.

Step 7: Adjusting the store

In order for us to be able to use asynchronous action creator functions in Redux, we are going to need to load the `redux-thunk` middleware. This middleware checks if an action creator returned a function, rather than a plain object, and if that is the case, it executes that function, while passing the `dispatch` function to it as an argument.

Let's adjust the store to allow for asynchronous action creators now:

1. Create a new `src/configureStore.js` file.
2. Import the `createStore` and `applyMiddleware` functions from Redux:

```
import { createStore, applyMiddleware } from 'redux'
```

3. Next, import the `thunk` middleware and `appReducer` function:

```
import thunk from 'redux-thunk'

import appReducer from './reducers'
```

4. Now, we can define the store and apply the `thunk` middleware to it:

```
const store = createStore(appReducer, applyMiddleware(thunk))
```

5. Finally, we export the `store`:

```
export default store
```

Using the `redux-thunk` middleware, we can now dispatch functions that will later dispatch actions, which means that our asynchronous action creator is going to work fine now.

Exercise 2: Adjusting reducers(Chapter12_2)

As previously mentioned, Redux reducers differ from Reducer Hooks in that they have certain conventions:

- Each reducer needs to set its initial state by defining a default value in the function definition
- Each reducer needs to return the current state for unhandled actions

We are now going to adjust our existing reducers so that they follow these conventions. The second convention is already implemented, because we defined a single app reducer earlier, in order to avoid having multiple dispatch functions.

Step 1: Setting the initial state in Redux reducers

So, we are going to focus on the first convention—to set the initial state by defining a default value in the function arguments, as follows:

1. Edit `src/reducers.js` and import the `combineReducers` function from Redux:

```
import { combineReducers } from 'redux'
```

2. Then, rename `filterReducer` to `filter`, and set a default value:

```
function filter (state = 'all', action) {
```

3. Next, edit `todosReducer` and repeat the same process there:

```
function todos (state = [], action) {
```

4. Finally, we are going to use the `combineReducers` function to create our `appReducer` function. Instead of creating the function manually, we can now do the following:

```
const appReducer = combineReducers({ todos, filter }) export
default appReducer
```

As we can see, Redux reducers are very similar to Reducer Hooks. Redux even provides a function that allows us to combine multiple reducer functions into a single app reducer!

Exercise 3: Connecting components

Now, it is time to introduce connectors and container components. In Redux we can use the `connect` higher-order component to connect existing components to Redux, through injecting state and action creators as props into them.

Redux defines two different kinds of components:

- **Presentational components**: React components, as we have been defining them until now
- **Container components**: React components that connect presentational components to Redux

Container components use a connector to connect Redux to a presentational component. This connector accepts two functions:

- `mapStateToProps(state)`: Takes the current Redux state, and returns an object of props to be passed to the component; used to pass state to the component
- `mapDispatchToProps(dispatch)`: Takes the `dispatch` function from the Redux store, and returns an object of props to be passed to the component; used to pass action creators to the component

Step 1: We are now going to define container components for our existing presentational components:

1. First, we create a new `src/components/` folder for all our presentational components.
2. Then, we copy all of the existing component files to the `src/components/` folder, and adjust the import statements for the following files:

   `AddTodo.js`, `App.js`, `Header.js`, `TodoFilter.js`, `TodoItem.js`, and `TodoList.js`.

Step 2: Connecting the AddTodo component

We are now going to start connecting our components to the Redux store. The presentational components can stay the same as before. We only create new components—container components—that wrap the presentational components, and pass certain props to them.

Let's connect the `AddTodo` component now:

1. Create a new `src/containers/` folder for all our container components.
2. Create a new `src/containers/ConnectedAddTodo.js` file.
3. In this file, we import the `connect` function from `react-redux`, and the `bindActionCreators` function from `redux`:

```
import { connect } from 'react-redux'
import { bindActionCreators } from 'redux'
```

4. Next, we import the `addTodo` action creator and the `AddTodo` component:

```
import { addTodo } from '../actions'
import AddTodo from '../components/AddTodo'
```

5. Now, we are going to define the `mapStateToProps` function. Since this component does not deal with any state from Redux, we can simply return an empty object here:

```
function mapStateToProps (state) {
    return {} }
```

6. Then, we define the `mapDispatchToProps` function. Here we use `bindActionCreators` to wrap the action creator with the `dispatch` function:

```
function mapDispatchToProps (dispatch) {
    return bindActionCreators({ addTodo }, dispatch)
}
```

   This code is essentially the same as manually wrapping the action creators, as follows:

```
function mapDispatchToProps (dispatch) {
    return {
        addTodo: (...args) => dispatch(addTodo(...args))
    }
}
```

7. Finally, we use the `connect` function to connect the `AddTodo` component to Redux:

```
export default connect(mapStateToProps, mapDispatchToProps)(AddTodo)
```

Now, our `AddTodo` component is successfully connected to the Redux store.

Step 3: Connecting the TodoItem component

Next, we are going to connect the `TodoItem` component, so that we can use it in the `TodoList` component in the next step.

Let's connect the `TodoItem` component now:

1. Create a new `src/containers/ConnectedTodoItem.js` file.
2. In this file, we import the `connect` function from `react-redux`, and the `bindActionCreators` function from `redux`:

```
import { connect } from 'react-redux'
import { bindActionCreators } from 'redux'
```

3. Next, we import the `toggleTodo` and `removeTodo` action creators, and the `TodoItem` component:

```
import { toggleTodo, removeTodo } from '../actions'
import TodoItem from '../components/TodoItem'
```

4. Again, we only return an empty object from `mapStateToProps`:

```
function mapStateToProps (state) {
    return {}
}
```

5. This time, we bind two action creators to the `dispatch` function:

```
function mapDispatchToProps (dispatch) {
    return bindActionCreators({ toggleTodo, removeTodo }, dispatch)
}
```

6. Finally, we connect the component, and export it:

```
export default connect(mapStateToProps, mapDispatchToProps)(TodoItem)
```

Now, our `TodoItem` component is successfully connected to the Redux store.

Step 4: Connecting the TodoList component

After connecting the `TodoItem` component, we can now use the `ConnectedTodoItem` component in the `TodoList` component.

Let's connect the `TodoList` component now:

1. Edit `src/components/TodoList.js`, and adjust the import statement as follows:

```
import ConnectedTodoItem from '../containers/ConnectedTodoItem'
```

2. Then, rename the component that is returned from the function to `ConnectedTodoItem`:

```
return filteredTodos.map(item =>
    <ConnectedTodoItem {...item} key={item.id} />
)
```

3. Now, create a new `src/containers/ConnectedTodoList.js` file.

4. In this file, we import only the `connect` function from `react-redux`, as we are not going to bind the action creators this time:

```
import { connect } from 'react-redux'
```

5. Next, we import the `TodoList` component:

```
import TodoList from '../components/TodoList'
```

6. Now, we define the `mapStateToProps` function. This time, we use destructuring to get `todos` and `filter` from the `state` object, and return them:

```
function mapStateToProps (state) {
    const { filter, todos } = state
    return { filter, todos } }
```

7. Next, we define the `mapDispatchToProps` function, where we only return an empty object, since we are not going to pass any action creators to the `TodoList` component:

```
function mapDispatchToProps (dispatch) {
    return {}
}
```

8. Finally, we connect and export the connected `TodoList` component:

```
export default connect(mapStateToProps, mapDispatchToProps)(TodoList)
```

Now, our `TodoList` component is successfully connected to the Redux store.

Step 5: Adjusting the TodoList component

Now that we have connected the `TodoList` component, we can move the filter logic from the `App` component to the `TodoList` component, as follows:

1. Import the `useMemo` Hook in `src/components/TodoList.js`:

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

2. Edit `src/components/App.js`, and remove the following code:

```
const filteredTodos = useMemo(() => {
    const { filter, todos } = state
    switch (filter) {
        case 'active':
            return todos.filter(t => t.completed === false)
        case 'completed':
            return todos.filter(t => t.completed === true)
        default:
        case 'all':
            return todos
    }
}, [ state ])
```

3. Now, edit `src/components/TodoList.js`, and add the `filteredTodos` code here. Please note that we removed the destructuring from the state object, as the component already receives the `filter` and `todos` values as props. We also adjusted the dependency array accordingly:

```
const filteredTodos = useMemo(() => {
    switch (filter) {
        case 'active':
            return todos.filter(t => t.completed === false)
        case 'completed':
            return todos.filter(t => t.completed === true)
        default:
        case 'all':
            return todos
    }
}, [ filter, todos ])
```

Now, our filtering logic is in the `TodoList` component, instead of the `App` component. Let's move on to connecting the rest of our components.

Step 6: Connecting the TodoFilter component

Next up is the `TodoFilter` component. Here, we are going to use both `mapStateToProps` and `mapDispatchToProps`.

Let's connect the `TodoFilter` component now:

1. Create a new `src/containers/ConnectedTodoFilter.js` file.
2. In this file, we import the `connect` function from `react-redux` and the `bindActionCreators` function from `redux`:

```
import { connect } from 'react-redux'
import { bindActionCreators } from 'redux'
```

3. Next, we import the `filterTodos` action creator and the `TodoFilter` component:

```
import { filterTodos } from '../actions'
import TodoFilter from '../components/TodoFilter'
```

4. We use destructuring to get the `filter` from our `state` object, and then we return it:

```
function mapStateToProps (state) {
    const { filter } = state
    return { filter }
}
```

5. Next, we bind and return the `filterTodos` action creator:

```
function mapDispatchToProps (dispatch) {
    return bindActionCreators({ filterTodos }, dispatch)
}
```

6. Finally, we connect the component and export it:

```
export default connect(mapStateToProps, mapDispatchToProps)(TodoFilter)
```

Now, our `TodoFilter` component is successfully connected to the Redux store.

Step 7: Connecting the App component

The only component that still needs to be connected now, is the `App` component. Here, we are going to inject the `fetchTodos` action creator, and update the component so that it uses the connected versions of all the other components.

Let's connect the `App` component now:

1. Edit `src/components/App.js`, and adjust the following import statements:

```
import ConnectedAddTodo from '../containers/ConnectedAddTodo'

import ConnectedTodoList from '../containers/ConnectedTodoList'

import ConnectedTodoFilter from '../containers/ConnectedTodoFilter'
```

2. Also, adjust the following components that are returned from the function:

```
return (
    <div style={{ width: 400 }}>
        <Header />
        <ConnectedAddTodo />
        <hr />
        <ConnectedTodoList />
        <hr />
        <ConnectedTodoFilter />
    </div>
)
```

3. Now, we can create the connected component. Create a new `src/containers/ConnectedApp.js` file.
4. In this newly created file, we import the `connect` function from `react-redux`, and the `bindActionCreators` function from `redux`:

```
import { connect } from 'react-redux'
import { bindActionCreators } from 'redux'
```

5. Next, we import the `fetchTodos` action creator, and the `App` component:

```
import { fetchTodos } from '../actions' import App
from '../components/App'
```

6. We already deal with the various parts of our state in other components, so there is no need to inject any state into our `App` component:

```
function mapStateToProps (state) {
    return {}
}
```

7. Then, we bind and return the `fetchTodos` action creator:

```
function mapDispatchToProps (dispatch) {
    return bindActionCreators({ fetchTodos }, dispatch)
}
```

8. Finally, we connect the `App` component and export it:

```
export default connect(mapStateToProps, mapDispatchToProps)(App)
```

Now, our `App` component is successfully connected to the Redux store.

Step 8: Setting up the Provider component

Finally, we have to set up a `Provider` component, which is going to provide a context for the Redux store, which will be used by the connectors.

Let's set up the `Provider` component now:

1. Edit `src/index.js`, and import the `Provider` component from `react-redux`:

```
import { Provider } from 'react-redux'
```

2. Now, import the `ConnectedApp` component from the `containers` folder and import the Redux store that was created by `configureStore.js`:

```
import ConnectedApp from './containers/ConnectedApp'

import store from './configureStore'
```

3. Finally, adjust the first argument to `ReactDOM.render`, by wrapping the `ConnectedApp` component with the `Provider` component, as follows:

```
ReactDOM.render(
    <Provider store={store}>
        <ConnectedApp />
    </Provider>,
    document.getElementById('root')
)
```

Now, our application will work in the same way as before, but everything is connected to the Redux store! As we can see, Redux requires a bit more boilerplate code than simply using React, but it comes with a lot of advantages:

- Easier handling of asynchronous actions (using the `redux-thunk` middleware)
- Centralized action handling (no need to define action creators in the components)
- Useful functions for binding action creators and combining reducers
- Reduced possibilities for errors (for example, by using action types, we can ensure that we did not make a typo)

However, there are also disadvantages, which are as follows:

- A lot of boilerplate code is required (action types, action creators, and connected components)
- Mapping of state/action creators in separate files (not in the components, where they are needed)

The first point is an advantage and disadvantage at the same time; action types and action creators do require more boilerplate code, but they also make it easier to update actionrelated code at a later stage. The second point, and the boilerplate code that is required for the connected components, can be solved by using Hooks to connect our components to Redux. We are going to use Hooks with Redux in the next section of this chapter.

Example code

The example code can be found in the `Chapter12/chapter12_2` folder.

Just run `npm install` in order to install all dependencies and `npm start` to start the application, then visit `http://localhost:3000` in your browser (if it did not open automatically).

Exercise 4: Using Redux with Hooks (Chapter12_3)

After turning our todo application into a Redux-based application, we are now using higher-order components, instead of Hooks, in order to get access to the Redux state and action creators. This is the traditional way to develop a Redux application. However, in the latest versions of Redux, it is possible to use Hooks instead of higher-order components!
We are now going to replace the existing connectors with Hooks.

> Even with Hooks, the `Provider` component is still required in order to provide the Redux store to other components. The definition of the store and the provider can stay the same when refactoring from `connect()` to Hooks.

The latest version of React Redux offers various Hooks as an alternative to the `connect()` higher-order component. With these Hooks, you can subscribe to the Redux store, and dispatch actions without having to wrap your components.

Using the dispatch Hook

The `useDispatch` Hook returns a reference to the `dispatch` function that is provided by the Redux store. It can be used to dispatch actions that are returned from action creators. Its API looks as follows: `const dispatch = useDispatch()`

We are now going to use the Dispatch Hook to replace the existing container components with Hooks.

> You do not need to migrate your whole Redux application at once in order to use Hooks. It is possible to selectively refactor certain components—meaning that they will use Hooks—while still using connect() for other components.

After learning how to use the Dispatch Hook, let's move on to migrating our existing components so that they use the Dispatch Hook.

Step 1: Using Hooks for the AddTodo component

Now that we have learned about the Dispatch Hook, let's see it in action by implementing it in our `AddTodo` component.

Let's migrate the `AddTodo` component to Hooks now:

1. First delete the `src/containers/ConnectedAddTodo.js` file.
2. Now, edit the `src/components/AddTodo.js` file and import the `useDispatch` Hook from `react-redux`:

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

3. Additionally, import the `addTodo` action creator:

```
import { addTodo } from '../actions'
```

4. Now, we can remove the props from the function definition:

```
export default function AddTodo () {
```

5. Then, define the Dispatch Hook:

```
const dispatch = useDispatch()
```

6. Finally, adjust the handler function and call `dispatch()`:

```
function handleAdd () {
    if (input) {
        dispatch(addTodo(input))
        setInput('')
    }
}
```

7. Now, all that is left to do is to replace the `ConnectedAddTodo` component with the `AddTodo` component in `src/components/App.js`. First, adjust the import statement:

```
import AddTodo from './AddTodo'
```

8. Then, adjust the rendered component:

```
return (
    <div style={{ width: 400 }}>
        <Header />
        <AddTodo />
```

As you can see, our app still works in the same way as before, but we are now using Hooks in order to connect the component to Redux!

Step 2: Using Hooks for the App component

Next, we are going to update our `App` component so that it directly dispatches the `fetchTodos` action. Let's migrate the `App` component to Hooks now:

1. First delete the `src/containers/ConnectedApp.js` file.
2. Now, edit the `src/components/App.js` file and import the `useDispatch` Hook from `react-redux`:

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

3. Additionally, import the `fetchTodos` action creator:

```
import { fetchTodos } from '../actions'
```

4. Now, we can remove the props from the function definition:

```
export default function App () {
```

5. Then, define the Dispatch Hook:

```
const dispatch = useDispatch()
```

6. Finally, adjust the Effect Hook and call `dispatch()`:

```
useEffect(() => {
    dispatch(fetchTodos())
}, [ dispatch ])
```

7. Now, all that is left to do is to replace the `ConnectedApp` component with the `App` component in `src/index.js`. First, adjust the import statement:

```
import App from './components/App'
```

8. Then, adjust the rendered component:

```
ReactDOM.render(
    <Provider store={store}>
        <App />
    </Provider>,
    document.getElementById('root') )
```

As we can see, using Hooks is much simpler and more concise than defining a separate container component.

Step 3: Using Hooks for the TodoItem component

Now, we are going to upgrade the `TodoItem` component to use Hooks. Let's migrate it now:

1. First delete the `src/containers/ConnectedTodoItem.js` file.
2. Now, edit the `src/components/TodoItem.js` file, and import the `useDispatch` Hook from `react-redux`:

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

3. Additionally, import the `toggleTodo` and `removeTodo` action creators:

```
import { toggleTodo, removeTodo } from '../actions'
```

4. Now, we can remove the action creator-related props from the function definition. The new code should look as follows:

```
export default function TodoItem ({ title, completed, id }) {
```

5. Then, define the Dispatch Hook:

```
const dispatch = useDispatch()
```

6. Finally, adjust the handler functions to call `dispatch()`:

```
function handleToggle () {
    dispatch(toggleTodo(id))
}

function handleRemove () {
    dispatch(removeTodo(id))
}
```

7. Now, all that is left to do is to replace the `ConnectedTodoItem` component with the `TodoItem` component in `src/components/TodoList.js`. First, adjust the import statement:

```
import TodoItem from './TodoItem'
```

8. Then, adjust the rendered component:

```
return filteredTodos.map(item =>
    <TodoItem {...item} key={item.id} />
)
```

Now the `TodoItem` component uses Hooks instead of a container component. Next, we are going to learn about the Selector Hook.

Exercise 5: Using the Selector Hook

Another very important Hook that is provided by Redux is the Selector Hook. It allows us to get data from the Redux store state, by defining a selector function. The API for this Hook is as follows:

```
const result = useSelector(selectorFn, equalityFn)
```

`selectorFn` is a function that works similarly to the `mapStateToProps` function. It will get the full state object as its only argument. The selector function gets executed whenever the component renders, and whenever an action is dispatched (and the state is different than the previous state).

It is important to note that returning an object with multiple parts of the state from one Selector Hook will force a re-render every time an action is dispatched. If multiple values from the store need to be requested, we can do the following:

- Use multiple Selector Hooks, each one returning a single field from the state object
- Use `reselect`, or a similar library, to create a memoized selector (we are going to cover this in the next section)
- Use the `shallowEqual` function from `react-redux` as `equalityFn`

We are now going to implement the Selector Hook in our ToDo application, specifically in the `TodoList` and `TodoFilter` components.

Step 1: Using Hooks for the TodoList component

First, we are going to implement a Selector Hook to get all `todos` for the `TodoList` component, as follows:

1. First delete the `src/containers/ConnectedTodoList.js` file.
2. Now, edit the `src/components/TodoList.js` file, and import the `useSelector` Hook from `react-redux`:

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

3. Now, we can remove all the props from the function definition:

```
export default function TodoList () {
```

4. Then, we define two Selector Hooks, one for the `filter` value, and one for the `todos` value:

```
const filter = useSelector(state => state.filter)
const todos = useSelector(state => state.todos)
```

5. Now, all that is left to do is to replace the `ConnectedTodoList` component with the `TodoList` component in `src/components/App.js`. First, adjust the import statement:

```
import TodoList from './TodoList'
```

6. Then, adjust the rendered component:

```
return (
    <div style={{ width: 400 }}>
        <Header />
        <AddTodo />
        <hr />
        <TodoList />
```

The rest of the component can stay the same, because the values where we store the parts of the state have the same names as before.

Step 2: Using Hooks for the TodoFilter component

Finally, we are going to implement both the Selector and Dispatch Hooks in the `TodoFilter` component, because we need to highlight the current filter (state from the Selector Hook) and dispatch an action to change the filter (the Dispatch Hook).

Let's implement Hooks for the `TodoFilter` component now:

1. First, delete the `src/containers/ConnectedTodoFilter.js` file.
2. We can also delete the `src/containers/` folder, as it is empty now.
3. Now, edit the `src/components/TodoFilter.js` file, and import the `useSelector` and `useDispatch` Hooks from `react-redux`:

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

4. Additionally, import the `filterTodos` action creator:

```
import { filterTodos } from '../actions'
```

5. Now, we can remove all the props from the function definition:

```
export default function TodoFilter () {
```

6. Then, define the Dispatch and Selector Hooks:

```
const dispatch = useDispatch()
const filter = useSelector(state => state.filter)
```

7. Finally, adjust the handler function to call `dispatch()`:

```
function handleFilter () {
    dispatch(filterTodos(name))
}
```

8. Now, all that is left to do is to replace the `ConnectedTodoFilter` component with the `TodoFilter` component in `src/components/App.js`. First, adjust the import statement:

```
import TodoFilter from './TodoFilter'
```

9. Then, adjust the rendered component:

```
return (
    <div style={{ width: 400 }}>
        <Header />
        <AddTodo />
        <hr />
        <TodoList />
        <hr />
        <TodoFilter />
    </div>
)
```

Exercise 6: (Chapter12_4)

Now, our Redux application makes full use of Hooks instead of container components!
When defining selectors as we have done until now, a new instance of the selector is created every time the component is rendered. This is fine, if the selector function does not do any complex operations and does not maintain internal state. Otherwise, we need to use reusable selectors, which we are going to learn about now.

Step 1: Setting up reselect

In order to create reusable selectors, we can use the `createSelector` function from the `reselect` library. First, we have to install the library via `npm`. Execute the following command:

```
> npm install --save reselect
```

Now, the `reselect` library has been installed, and we can use it to create reusable selectors.

Step 2: Memoizing selectors that only depend on state

If we want to memoize selectors, and the selector only depends on the state (not props), we can declare the selector outside of the component, as follows:

1. Edit the `src/components/TodoList.js` file, and import the `createSelector` function from `reselect`:

   ```
   import { createSelector } from 'reselect'
   ```

2. Then, we define selectors for the `todos` and `filter` parts of the state, before the component definition:

   ```
   const todosSelector = state => state.todos
   const filterSelector = state => state.filter
   ```

   If selectors are used by many components, it might make sense to put them in a separate selectors.js file, and import them from there. For example, we could put the filterSelector in a separate file, and then import it in TodoList.js, as well as TodoFilter.js.

3. Now, we define a selector for the filtered todos, before the component is defined, as follows:

   ```
   const selectFilteredTodos = createSelector(
   ```

4. First, we specify the other two selectors that we want to reuse:

   ```
   todosSelector,
   filterSelector,
   ```

5. Now, we specify a filtering selector, copying the code from the `useMemo` Hook:

```
        (todos, filter) => {
            switch (filter) {
                case 'active':
                    return todos.filter(t => t.completed === false)
                case 'completed':
                    return todos.filter(t => t.completed === true)
                default:
                case 'all':
                    return todos
            }
        }
    )
```

6. Finally, we use our defined selector in the Selector Hook:

```
    export default function TodoList () {
        const filteredTodos = useSelector(selectFilteredTodos)
```

Now that we have defined a reusable selector for the filtered todos, the result of filtering the todos will be memoized, and will not be re-computed if the state did not change.