13 MobX and Hooks

Exercise 1: Handling state with MobX

The best way to learn about MobX is by using it in practice and seeing how it works. So, let's start by porting our ToDo application from `Chapter 11`, *Migrating from React Class Components*, to MobX. We start by copying the code example from `Chapter11/chapter11_2/`.

Step 1: Installing MobX

The first step is to install MobX and MobX React, via `npm`. Execute the following command:

```
> npm install --save mobx mobx-react
```

Now that MobX and MobX React are installed, we can start setting up the store.

Step 2: Setting up the MobX store

After installing MobX, it is time to set up our MobX store. The store will store all state, and the related computed values and actions. It is usually defined with a class.

Let's define the MobX store now:

1. Create a new `src/store.js` file.
2. Import the `observable`, `action`, and `computed` decorators, as well as the `decorate` function from MobX. These will be used to tag various functions and values in our store:

```
import { observable, action, computed, decorate } from 'mobx'
```

3. Also import the `fetchAPITodos` and `generateID` functions from our API code:

```
import { fetchAPITodos, generateID } from './api'
```

4. Now, we define the store by using a class:

```
export default class TodoStore {
```

5. In this store, we store a `todos` array, and the `filter` string value. These two values are observables. We are going to tag them as such later on:

```
todos = []
filter = 'all'
```

With a special project setup, we could use an experimental JavaScript feature, known as decorators, to tag our values as observables by writing @observable todos = []. However, this syntax is not supported by create-react-app, since it is not part of the JavaScript standard yet.

6. Next, we define a computed value in order to get all of the filtered `todos` from our store. The function will be similar to the one that we had in `src/App.js`, but now we will use `this.filter` and `this.todos`. Again, we have to tag the function as `computed` later on. MobX will automatically trigger this function when needed, and store the result until the state that it depends on changes:

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

7. Now, we define our actions. We start with the `fetch` action. As before, we have to tag our action functions with the `action` decorator at a later point. In MobX, we can directly modify our state by setting `this.todos`. Because the `todos` value is observable, any changes to it will be automatically tracked by MobX:

```
fetch () {
    fetchAPITodos().then((fetchedTodos) => {
        this.todos = fetchedTodos
    })
}
```

8. Then, we define our `addTodo` action. In MobX, we do not use immutable values, so we should not create a new array. Instead, we always modify the existing `this.todos` value:

```
addTodo (title) {
    this.todos.push({ id: generateID(), title, completed: false})
}
```

As you can see, MobX takes a more imperative approach, where values are directly modified, and MobX automatically keeps track of the changes. We do not need to use the rest/spread syntax to create new arrays; instead, we modify the existing state array directly.

9. Next up is the `toggleTodo` action. Here, we loop through all of the `todos` and modify the item with a matching `id`. Note how we can modify items within an array, and the change will still be tracked by MobX. In fact, MobX will even notice that only one value of the array has changed. In combination with React, this means that the list component will not re-render; only the item component of the item that changed is going to re-render. Please note that for this to be possible, we have to split up our components appropriately, such as making separate list and item components:

```
toggleTodo (id) {
    for (let todo of this.todos) {
        if (todo.id === id) {
            todo.completed = !todo.completed
            break
        }
    }
}
```

The for (let .. of ..) { construct will loop through all items of an array, or any other iterable value.

10. Now, we define the `removeTodo` action. First, we find the `index` of the `todo` item that we want to remove:

```
    removeTodo (id) {
        let index = 0
        for (let todo of this.todos) {
            if (todo.id === id) {
               break
            } else {
                index++
            }
        }
```

11. Then, we use `splice` to remove one element—starting from the `index` of the found element. This means that we cut out the item with the given `id` from our array:

```
        this.todos.splice(index, 1)
    }
```

12. The last action that we define, is the `filterTodos` action. Here, we simply set the `this.filter` value to the new filter:

```
    filterTodos (filterName) {
        this.filter = filterName
    }
}
```

13. Finally, we have to decorate our store with the various decorators that we mentioned earlier. We do this by calling the `decorate` function on our store class and passing an object mapping values and methods to decorators:

```
decorate(TodoStore, {
```

14. We start with the `todos` and `filter` values, which are observables:

```
    todos: observable,
    filter: observable,
```

15. Then, we decorate the `computed` value—`filteredTodos`:

```
    filteredTodos: computed,
```

16. Last but not least, we decorate our actions:

```
    fetch: action,
    addTodo: action,
    toggleTodo: action,
    removeTodo: action,
    filterTodos: action })
```

Now, our MobX store is decorated properly and ready to be used!

Exercise 2: Defining the Provider component

We could now initialize the store in our `App` component, and pass it down to all of the other components. However, it is a better idea to use React Context. That way, we can access the store from anywhere in our app. MobX React offers a `Provider` component, which provides the store in a context.

Step 1: Let's get started using the `Provider` component now:

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

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

2. Then, import the `TodoStore` from our `store.js` file:

   ```
   import TodoStore from './store'
   ```

3. Now, we create a new instance of the `TodoStore` class:

   ```
   const store = new TodoStore()
   ```

4. Finally, we have to adjust the first argument to `ReactDOM.render()`, in order to wrap the `App` component with the `Provider` component:

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

   Unlike Redux, with MobX, it is possible to provide multiple stores in our app. However, here, we only provide one store, and we call it todoStore.

Now, our store is initialized and ready to be used in all other components.

Exercise 3: Connecting components

Now that our MobX store is available as a context, we can start connecting our components to it. To do so, MobX React provides the `inject` higher-order component, which we can use to inject the store into our components.
In this section, we are going to connect the following components to our MobX store:

- `App`
- `TodoList`
- `TodoItem`
- `AddTodo`
- `TodoFilter`

Step 1: Connecting the App component

We are going to start by connecting our `App` component, where we will use the `fetch` action to fetch all `todos` from our API, when the app initializes.
Let's connect the `App` component now:

1. Edit `src/App.js`, and import the `inject` function from `mobx-react`:

```
import { inject } from 'mobx-react'
```

2. Then, wrap the `App` component with `inject`. The `inject` function is used to inject the store (or multiple stores) as props to the component:

```
export default inject('todoStore')(function App ({ todoStore }) {
```

It is possible to specify multiple stores in the inject function, as follows: **inject('todoStore', 'otherStore').**
Then, two props will be injected: todoStore and otherStore.

3. Now that we have the `todoStore` available, we can use it to call the `fetch` action within our Effect Hook:

```
useEffect(() => {
    todoStore.fetch()
}, [ todoStore ])
```

4. We can now remove the `filteredTodos` Memo Hook, the handler functions, the `StateContext.Provider` component, and all of the props that we passed down to the other components:

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

Now, our `App` component will fetch `todos` from the API, and then they will be stored in the `TodoStore`.

Step 2: Connecting the TodoList component

After storing the `todos` in our store, we can get them from the store, and then we can list all of the todo items in the `TodoList` component.

Let's connect the `TodoList` component now:

1. Edit `src/TodoList.js` and import the `inject` and `observer` functions:

```
import { inject, observer } from 'mobx-react'
```

2. Remove all context-related imports and Hooks.
3. As before, we use the `inject` function to wrap the component. Additionally, we now wrap our component with the `observer` function. The `observer` function tells MobX that this component should re-render when the store updates:

```
export default inject('todoStore')(observer(function TodoList ({ todoStore
}) {
```

4. We can now use the `filteredTodos` computed value from our store, to list all todo items with the filter applied. To make sure that MobX can still track when changes to the `item` object occur, we *do not* use the spread syntax here. If we used the spread syntax, all of the todo items would re-render, even if only one changed:

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

Now, our app will already list all of the todo items. However, we cannot toggle or remove the todo items yet.

Step 3: Connecting the TodoItem component

To be able to toggle or remove todo items, we have to connect the `TodoItem` component. We also define the `TodoItem` component as an observer, so that MobX knows it will have to re-render the component when the `item` object changes.

Let's connect the `TodoItem` component now:

1. Edit `src/TodoItem.js`, and import the `inject` and `observer` functions from `mobx-react`:

```
    import { inject, observer } from 'mobx-react'
```

2. Then, wrap the `TodoItem` component with `inject` and `observer`:

```
export default inject('todoStore')(observer(function TodoItem ({ item,
todoStore }) {
```

3. We can now use destructuring of the `item` object within the component. As it is defined as an observer, MobX will be able to track changes to the `item` object, even after destructuring:

```
        const { title, completed, id } = item
```

4. Now that we have the `todoStore` available, we can use it to adjust our handler functions, and to call the corresponding actions:

```
function handleToggle () {
    todoStore.toggleTodo(id)
}

function handleRemove () {
    todoStore.removeTodo(id)
}
```

Now, our `TodoItem` component will call the `toggleTodo` and `removeTodo` actions from our `todoStore`, so we can now toggle and remove the todo items!

Step 4: Connecting the AddTodo component

To be able to add new todo items, we have to connect the `AddTodo` component.

Let's connect the `AddTodo` component now:

1. Edit `src/AddTodo.js` and import the `inject` function from `mobx-react`:

```
import { inject } from 'mobx-react'
```

2. Then, wrap the `AddTodo` component with `inject`:

```
export default inject('todoStore')(function AddTodo ({ todoStore }) {
```

3. Now that we have the `todoStore` available, we can use it to adjust our handler function, and to call the `addTodo` action:

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

Now, our `AddTodo` component will call the `addTodo` action from our `todoStore`, so we can now add new todo items!

Step 5: Connecting the TodoFilter component

Lastly, we have to connect the `TodoFilter` component in order to be able to select different filters. We also want to show the currently selected filter, so this component needs to be an `observer`.

Let's connect the `TodoFilter` component now:

1. Edit `src/TodoFilter.js` and import the `inject` and `observer` functions:

```
import { inject, observer } from 'mobx-react'
```

2. We use the `inject` and `observer` functions to wrap the component:

```
const TodoFilterItem = inject('todoStore')(observer(function
TodoFilterItemWrapped ({ name, todoStore }) {
```

3. We now adjust our handler function to call the `filterTodos` action from the store:

```
function handleFilter () {
    todoStore.filterTodos(name)
}
```

4. Finally, we adjust the `style` object to use the `filter` value from `todoStore`, in order to check whether the filter is currently selected:

```
const style = {
    color: 'blue',
    cursor: 'pointer',
    fontWeight: (todoStore.filter === name) ? 'bold': 'normal'
}
```

5. Furthermore, we can now get rid of passing down the props in the `FilterItem` component. Remove the following parts that are marked in bold:

```
export default function TodoFilter (props) {
    return (
        <div>
            <TodoFilterItem {...props} name="all" />{' / '}
            <TodoFilterItem {...props} name="active" />{' / '}
            <TodoFilterItem {...props} name="completed" />
        </div>
    )
}
```

Now, we can select new filters, which will be marked as selected, in bold. The todo list will also automatically be filtered, because MobX detects a change in the `filter` value, which causes the `filteredTodos` computed value to update, and the `TodoList` observer component to re-render.

Exercise 4: Using MobX with Hooks (Chapter13_2)

In the previous section, we learned how to use MobX with React. As we have seen, to be able to connect our components to the MobX store, we need to wrap them with the `inject` function, and in some cases, also with the `observer` function. Instead of using these higher-order components to wrap our components, since the release of v6 of `mobx-react`, we can also use Hooks to connect our components to the MobX store. We are now going to use MobX with Hooks!

Step 1: Defining a store Hook

First of all, we have to define a Hook in order to access our own store. As we have learned before, MobX uses React Context to provide, and inject, state into various components. We can get the `MobXProviderContext` from `mobx-react` and create our own custom context Hook in order to access all stores. Then, we can create another Hook, to specifically access our `TodoStore`.

So, let's begin defining a store Hook:

1. Create a new `src/hooks.js` file.
2. Import the `useContext` Hook from `react`, and the `MobXProviderContext` from `mobx-react`:

```
import { useContext } from 'react'
import { MobXProviderContext } from 'mobx-react'
```

3. Now, we define and export a `useStores` Hook, which returns a Context Hook for the `MobXProviderContext`:

```
export function useStores () {
    return useContext(MobXProviderContext) }
```

4. Finally, we define a `useTodoStore` Hook, which gets the `todoStore` from our previous Hook, and then returns it:

```
export function useTodoStore () {
    const { todoStore } = useStores()
    return todoStore
}
```

Now, we have a general Hook, to access all stores from MobX, and a specific Hook to access the `TodoStore`. If we need to, we can also define more Hooks for other stores at a later point.

Upgrading components to Hooks

After creating a Hook to access our store, we can use it instead of wrapping our components with the `inject` higher-order component function. In the upcoming sections, we will see how we can use Hooks to upgrade our various components.

Step 2: Using Hooks for the App component

We are going to start by upgrading our `App` component. It is possible to gradually refactor components so that they use Hooks instead. We do not need to refactor every component at once.

Let's use Hooks for the `App` component now:

1. Edit `src/App.js` and remove the following `import` statement:

```
import { inject } from 'mobx-react'
```

2. Then, import the `useTodoStore` Hook from our `hooks.js` file:

```
import { useTodoStore } from './hooks'
```

3. Now, remove the `inject` function that is wrapping the `App` component, and remove all props. The `App` function definition should now look as follows:

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

4. Finally, use our Todo Store Hook to get the `todoStore` object:

```
const todoStore = useTodoStore()
```

As you can see, our app still works in the same way as before! However, we are now using Hooks in the `App` component, which makes the code much more clean and concise.

Step 3: Using Hooks for the TodoList component

Next, we are going to upgrade our `TodoList` component. Additionally, we are also going to use the `useObserver` Hook, which replaces the `observer` higher-order component.

Let's use Hooks for the `TodoList` component now:

1. Edit `src/TodoList.js`, and remove the following import statement:

```
import { inject, observer } from 'mobx-react'
```

2. Then, import the `useObserver` Hook from `mobx-react` and the `useTodoStore` Hook from our `hooks.js` file:

```
import { useObserver } from 'mobx-react'

import { useTodoStore } from './hooks'
```

3. Now, remove the `inject` and `observer` functions that are wrapping the `TodoList` component, and also remove all props. The `TodoList` function definition should now look as follows:

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

4. Again, we use the Todo Store Hook to get the `todoStore` object:

```
const todoStore = useTodoStore()
```

5. Finally, we wrap the returned elements with the `useObserver` Hook. Everything within the Observer Hook will be recomputed when the state that is used within the Hook changes:

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

In our case, MobX will detect that the observer that was defined via the `useObserver` Hook depends on `todoStore.filteredTodos`, and `filteredTodos` depends on the `filter` and `todos` values. As a result, the list will be re-rendered whenever either the `filter` value or the `todos` array changes.

Step 4: Using Hooks for the TodoItem component

Next, we are going to upgrade the `TodoItem` component, which will be a similar process to what we did with the `TodoList` component.

Let's use Hooks for the `TodoItem` component now:

1. Edit `src/TodoItem.js` and remove the following `import` statement:

```
import { inject, observer } from 'mobx-react'
```

2. Then, import the `useObserver` Hook from `mobx-react`, and the `useTodoStore` Hook from our `hooks.js` file:

```
import { useObserver } from 'mobx-react'
import { useTodoStore } from './hooks'
```

3. Now, remove the `inject` and `observer` functions that are wrapping the `TodoItem` component, and also remove the `todoStore` prop. The `TodoItem` function definition should now look as follows:

```
export default function TodoItem ({ item }) {
```

4. Next, we have to remove the destructuring (the code in bold) because our whole component is not defined as observable anymore, so MobX will not be able to track the changes to the `item` object:

```
    const { title, completed, id } = item
```

5. Then, use the Todo Store Hook to get the `todoStore` object:

```
    const todoStore = useTodoStore()
```

6. Now, we have to adjust the handler functions so that they use `item.id` instead of `id` directly. Please note that we assume that the `id` does not change, therefore, it is not wrapped within an Observer Hook:

```
    function handleToggle () {
        todoStore.toggleTodo(item.id)
    }

    function handleRemove () {
        todoStore.removeTodo(item.id)
    }
```

7. Finally, we wrap the `return` statement with an Observer Hook and do the destructuring there. This ensures that changes to the `item` object are tracked by MobX, and that the component will re-render accordingly when the properties of the object change:

```
    return useObserver(() => {
        const { title, completed } = item
        return (
            <div style={{ width: 400, height: 25 }}>
                <input type="checkbox" checked={completed}
onChange={handleToggle} />
                {title}
                <button style={{ float: 'right' }}
onClick={handleRemove}>x</button>
            </div>
        )
    })
}
```

Now, our `TodoItem` component is properly connected to the MobX store.

If the `item.id` property changes, we would have to wrap the handler functions, and the `return` function, within a single `useObserver` Hook, as follows:

```
    return useObserver(() => {
        const { title, completed, id } = item

        function handleToggle () {
            todoStore.toggleTodo(id)
        }

        function handleRemove () {
            todoStore.removeTodo(id)
        }

        return (
            <div style={{ width: 400, height: 25 }}>
                <input type="checkbox" checked={completed}
onChange={handleToggle} />
                {title}
                <button style={{ float: 'right' }}
onClick={handleRemove}>x</button>
            </div>
        )
    })
```

Note that we cannot wrap the handler functions and the `return` statement in separate Observer Hooks, because then the handler functions would only be defined within the closure of the first Observer Hook. This would mean that we would not be able to access the handler functions from within the second Observer Hook.

Next, we are going to continue to upgrade our components by using Hooks for the `AddTodo` component.

Step 5: Using Hooks for the AddTodo component

We repeat the same upgrade process as we did in the `App` component for the `AddTodo` component, as follows:

1. Edit `src/AddTodo.js` and remove the following `import` statement:

```
import { inject } from 'mobx-react'
```

2. Then, import the `useTodoStore` Hook from our `hooks.js` file:

```
import { useTodoStore } from './hooks'
```

3. Now, remove the `inject` function that is wrapping the `AddTodo` component, and also remove all props. The `AddTodo` function definition should now look as follows:

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

4. Finally, use the Todo Store Hook to get the `todoStore` object:

```
const todoStore = useTodoStore()
```

Now, our `AddTodo` component is connected to the MobX store and we can move on to upgrading the `TodoFilter` component.

Step 6: Using Hooks for the TodoFilter component

For the `TodoFilter` component, we are going to use a similar process to the one that we used for the `TodoList` component. We are going to use our `useTodoStore` Hook and the `useObserver` Hook.

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

1. Edit `src/TodoFilter.js` and remove the following `import` statement:

```
import { inject, observer } from 'mobx-react'
```

2. Then, import the `useObserver` Hook from `mobx-react`, and the `useTodoStore` Hook from our `hooks.js` file:

```
import { useObserver } from 'mobx-react'
import { useTodoStore } from './hooks'
```

3. Now, remove the `inject` and `observer` functions that are wrapping the `TodoFilterItem` component, and also remove the `todoStore` prop. The `TodoFilterItem` function definition should now look as follows:

```
function TodoFilterItem ({ name }) {
```

4. Again, we use the Todo Store Hook to get the `todoStore` object:

```
const todoStore = useTodoStore()
```

5. Finally, we wrap the `style` object with the `useObserver` Hook. Remember, everything within the Observer Hook will be re-computed when the state that is used within the Hook changes:

```
const style = useObserver(() => ({
    color: 'blue',
    cursor: 'pointer',
    fontWeight: (todoStore.filter === name) ? 'bold' : 'normal'
}))
```

In this case, the `style` object will be re-computed whenever the `todoStore.filter` value changes, which will cause the element to re-render, and change the font weight when a different filter is selected.

Step 7: (Chapter13_3)Using the local store Hook

In addition to providing global stores to store application-wide state, MobX also provides local stores to store local state. To create a local store, we can use the `useLocalStore` Hook.

We are now going to implement the Local Store Hook in the `AddTodo` component:

1. Edit `src/AddTodo.js` and import the `useLocalStore` Hook, as well as the `useObserver` Hook from `mobx-react`:

```
import { useLocalStore, useObserver } from 'mobx-react'
```

2. Then, remove the following State Hook:

```
const [ input, setInput ] = useState('')
```

   Replace it with a Local Store Hook:

```
const inputStore = useLocalStore(() => ({
```

   In this local store, we can define state values, computed values, and actions. The `useLocalStore` Hook will automatically decorate values as observable, getter functions (the `get` prefix) as computed values, and normal functions as actions.

3. We start with a `value` state for the `input` field:

```
value: '',
```

4. Then, we define a computed value, which will tell us whether the **add** button should be `disabled` or not:

```
get disabled () {
    return !this.value
},
```

5. Next, we define the actions. The first action updates the `value` from an input event:

```
updateFromInput (e) {
    this.value = e.target.value
},
```

6. Then, we define another action to update the `value` from a simple string:

```
        update (val) {
            this.value = val
        }
    }))
```

7. Now, we can adjust the input handler function, and call the `updateFromInput` action:

```
function handleInput (e) {
    inputStore.updateFromInput(e)
}
```

8. We also have to adjust the `handleAdd` function:

```
function handleAdd () {
    if (inputStore.value) {
        todoStore.addTodo(inputStore.value)
        inputStore.update('')
    }
}
```

9. Finally, we wrap the elements with a `useObserver` Hook, in order to make sure that the `input` field value gets updated when it changes, and we adjust the `disabled` and `value` props:

```
    return useObserver(() => (
        <form onSubmit={e => { e.preventDefault(); handleAdd() }}>
            <input
                type="text"
                placeholder="enter new task..."
                style={{ width: 350, height: 15 }}
                value={inputStore.value}
                onChange={handleInput}
            />
            <input
                type="submit"
                style={{ float: 'right', marginTop: 2 }}
                disabled={inputStore.disabled}
                value="add"
            />
        </form>
    ))
}
```

Now, our `AddTodo` component uses a local MobX store in order to handle its input value, and to disable/enable the button. As you can see, with MobX, it is possible to use multiple stores, for local as well as global states. The hard part is deciding how to split up and group your stores in a way that makes sense for the given application.