11 Migrating from React Class Components

Exercise 1: Handling state with class components

Before we start migrating from class components to Hooks, we are going to create a small ToDo list app using React class components. In the next section, we are going to turn these class components into function components using Hooks. Finally, we are going to compare the two solutions.

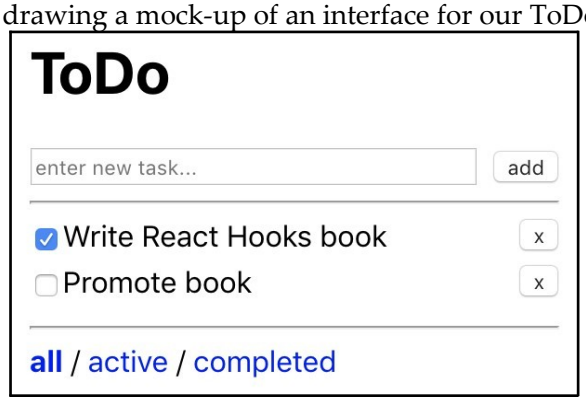
Designing the app structure

As we did before with the blog app, we are going to start by thinking about the basic structure of our app. For this app, we are going to need the following features:

- A header
- A way to add new todo items
- A way to show all todo items in a list
- A filter for the todo items

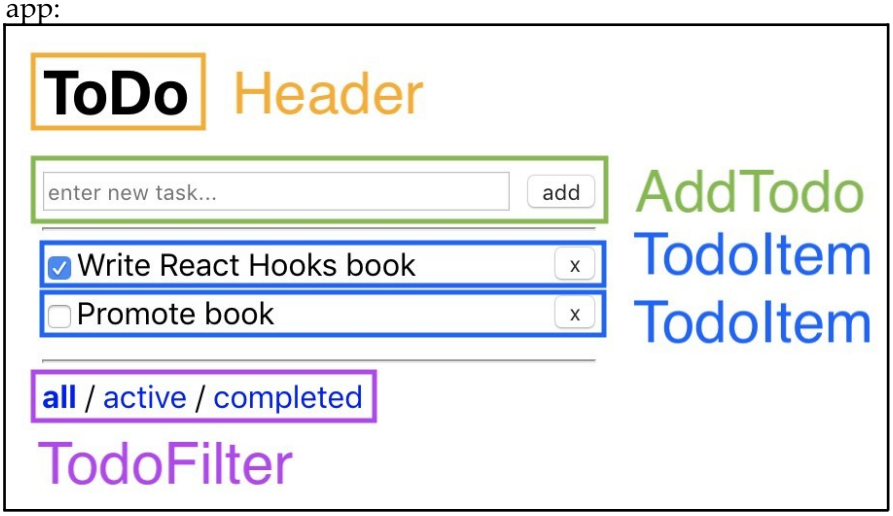
Step 1: It is always a good idea to start with a mock-up. So, let's begin:

1. We start by drawing a mock-up of an interface for our ToDo app:



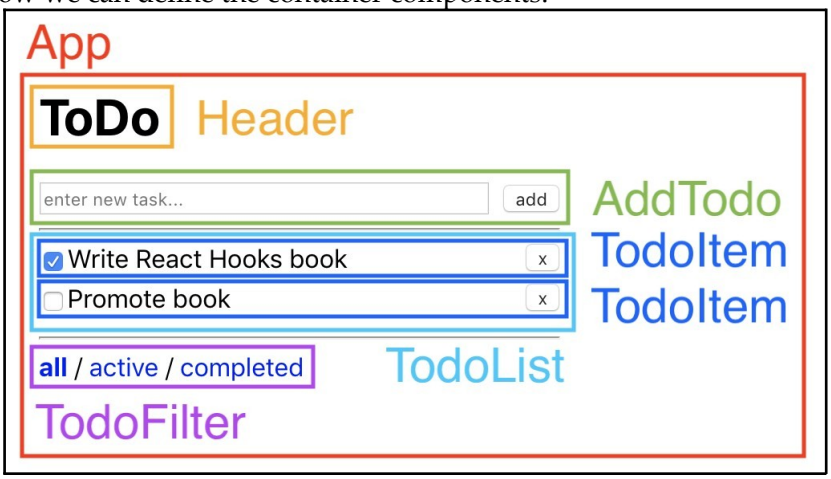
Mock-up of our ToDo app

2. Next, we define the fundamental components, in a similar way to how we did it with the blog app:



Defining fundamental components in our app mock-up

3. Now we can define the container components:



Defining container components in our app mock-up

As we can see, we are going to need the following components:

- `App`
- `Header`
- `AddTodo`
- `TodoList`
- `TodoItem`
- `TodoFilter (+ TodoFilterItem)`

The `TodoList` component makes use of a `TodoItem` component, which is used to show an item, with a checkbox to complete and a button to remove it. The `TodoFilter` component internally uses a `TodoFilterItem` component to show the various filters.

Step 2: Initializing the project

We are going to use `create-react-app` in order to create a new project. Let's initialize the project now:

1. Run the following command:

   ```
   > npx create-react-app chapter11_1
   ```

2. Then, remove `src/App.css`, as we are not going to need it.
3. Next, edit `src/index.css`, and adjust the margin as follows:

   ```
   margin: 20px;
   ```

4. Finally, remove the current `src/App.js` file, as we are going to create a new one in the next step.

Now, our project has been initialized, and we can start defining the app structure.

Step 3: Defining the app structure

We already know what the basic structure of our app is going to be like from the mock-up, so let's start by defining the `App` component:

1. Create a new `src/App.js` file.
2. Import `React` and the `Header`, `AddTodo`, `TodoList`, and `TodoFilter` components:

   ```
   import React from 'react'
   import Header from './Header'
   import AddTodo from './AddTodo'

   import TodoList from './TodoList'

   import TodoFilter from './TodoFilter'
   ```

3. Now define the `App` component as a class component. For now, we are only going to define the `render` method:

   ```
   export default class App extends React.Component {
       render () {
           return (
               <div style={{ width: 400 }}>
                   <Header />
                   <AddTodo />
                   <hr />
                   <TodoList />
                   <hr />
                   <TodoFilter />
               </div>
           )
       }
   }
   ```

The `App` component defines the basic structure of our app. It will consist of a header, a way to add new todo items, a list of todo items, and a filter.

Exercise 2: Defining the components

Now, we are going to define the components as static components. Later in this chapter, we are going to implement dynamic functionality to them. For now, we are going to implement the following static components:

- `Header`
- `AddTodo`
- `TodoList`
- `TodoItem`
- `TodoFilter`

Let's get started implementing the components now.

Step 1: Defining the Header component

We are going to start with the `Header` component, as it is the most simple out of all the components:

1. Create a new `src/Header.js` file.
2. Import `React` and define the class component with a `render` method: `import React from 'react'`

```
export default class Header extends React.Component {
    render () {
        return <h1>ToDo</h1>
    }
}
```

Now, the `Header` component for our app is defined.

Step 2: Defining the AddTodo component

Next, we are going to define the `AddTodo` component, which renders an `input` field and a button.

Let's implement the `AddTodo` component now:

1. Create a new `src/AddTodo.js` file.
2. Import `React` and define the class component and a `render` method: `import React from 'react'`

```
export default class AddTodo extends React.Component {
    render () {
        return (
```

3. In the `render` method, we return a `form` that contains an `input` field and an **add** button:

```
            <form>
                <input type="text" placeholder="enter new task..."
style={{ width: 350, height: 15 }} />
                <input type="submit" style={{ float: 'right',
marginTop: 2 }} value="add" />
            </form>
        )
    }
}
```

As we can see, the `AddTodo` component consists of an `input` field and a button.

Step 3: Defining the TodoList component

Now, we define the `TodoList` component, which is going to make use of the `TodoItem` component. For now, we are going to statically define two todo items in this component.

Let's start defining the `TodoList` component:

1. Create a new `src/TodoList.js` file.
2. Import `React` and the `TodoItem` component:

```
import React from 'react'
import TodoItem from './TodoItem'
```

3. Then, define the class component and a `render` method:

```
export default class TodoList extends React.Component {
    render () {
```

4. In this `render` method, we statically define two todo items:

```
        const items = [
            { id: 1, title: 'Write React Hooks book', completed: true },
            { id: 2, title: 'Promote book', completed: false }
        ]
```

5. Finally, we are going to render the items using the `map` function:

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

As we can see, the `TodoList` component renders a list of `TodoItem` components.

Step 4: Defining the TodoItem component

After defining the `TodoList` component, we are now going to define the `TodoItem` component, in order to render single items.

Let's start defining the `TodoItem` component:

1. Create a new `src/TodoItem.js` component.
2. Import `React`, and define the component, as well as the `render` method: `import React from 'react'`

```
export default class TodoItem extends React.Component {
    render () {
```

3. Now, we are going to use destructuring in order to get the `title` and `completed` props:

```
        const { title, completed } = this.props
```

4. Finally, we are going to render a `div` element containing a `checkbox`, a `title`, and a `button` to delete the item:

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

The `TodoItem` component consists of a checkbox, a `title`, and a `button` to delete the item.

Step 5: Defining the TodoFilter component

Finally, we are going to define the `TodoFilter` component. In the same file, we are going to define another component for the `TodoFilterItem`.

Let's start defining the `TodoFilterItem` and `TodoFilter` components:

1. Create a new `src/TodoFilter.js` file.
2. Define a class component for the `TodoFilterItem`:

```
class TodoFilterItem extends React.Component {
    render () {
```

3. In this `render` method, we use destructuring in order to get the `name` prop:

```
        const { name } = this.props
```

4. Next, we are going to define an object for the `style`:

```
            const style = {
            color: 'blue',
            cursor: 'pointer'         }
```

5. Then, we return a `span` element with the `name` value of the filter, and use the defined `style` object:

```
        return <span style={style}>{name}</span>
    }
}
```

6. Finally, we can define the actual `TodoFilter` component, which is going to render three `TodoFilterItem` components, as follows:

```
export default class TodoFilter extends React.Component {
    render () {
        return (
            <div>
                <TodoFilterItem name="all" />{' / '}
                <TodoFilterItem name="active" />{' / '}
                <TodoFilterItem name="completed" />
            </div>
        )
    }
}
```

Now, we have a component that lists the three different filter possibilities: `all`, `active`, and `completed`.

Exercise 3: Implementing dynamic code

Now that we have defined all of the static components, our app should look just like the mock-up. The next step is to implement dynamic code using React state, life cycle, and handler methods.
In this section, we are going to do the following:

- Define a mock API
- Define a `StateContext`
- Make the `App` component dynamic
- Make the `AddTodo` component dynamic
- Make the `TodoList` component dynamic
- Make the `TodoItem` component dynamic
- Make the `TodoFilter` component dynamic

Let's get started.

Step 1: Defining the API code

First of all, we are going to define an API that will fetch todo items. In our case, we are simply going to return an array of todo items, after a short delay.

Let's start implementing the mock API:

1. Create a new `src/api.js` file.
2. We are going to define a function that will generate a random ID for our todo items based on the **Universally Unique Identifier** (**UUID**) function:

```
export const generateID = () => {
    const S4 = ()
=>(((1+Math.random())*0x10000)|0).toString(16).substring(1)
    return (S4()+S4()+"-"+S4()+"-"+S4()+"-"+S4()+"-"+S4()+S4()+S4())
}
```

3. Then, we define the `fetchAPITodos` function, which returns a `Promise`, which resolves after a short delay:

```
export const fetchAPITodos = () =>
    new Promise((resolve) =>
        setTimeout(() => resolve([
            { id: generateID(), title: 'Write React Hooks book',
completed: true },
            { id: generateID(), title: 'Promote book', completed: false }
        ]), 100)
    )
```

Now, we have a function that simulates fetching todo items from an API, by returning an array after a delay of `100` ms.

Step 2: Defining the StateContext

Next, we are going to define a context that will keep our current list of todo items. We are going to call this context `StateContext`.

Let's start implementing the `StateContext` now:

1. Create a new `src/StateContext.js` file.
2. Import `React`, as follows:

```
import React from 'react'
```

3. Now, define the `StateContext` and set an empty array as the fallback value:

```
const StateContext = React.createContext([])
```

4. Finally, export the `StateContext`:

```
export default StateContext
```

Now, we have a context where we can store our array of todo items.

Step 3: Making the App component dynamic

We are now going to make the `App` component dynamic by adding functionality to fetch, add, toggle, filter, and remove todo items. Furthermore, we are going to define a `StateContext` provider.

Let's start making the `App` component dynamic:

1. In `src/App.js`, import the `StateContext`, after the other import statements:

```
import StateContext from './StateContext'
```

2. Then, import the `fetchAPITodos` and `generateID` functions from the `src/api.js` file:

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

3. Next, we are going to modify our `App` class code, implementing a `constructor`, which will set the initial state:

```
export default class App extends React.Component {

    constructor (props) {
```

4. In this `constructor`, we need to first call `super`, to make sure that the parent class (`React.Component`) constructor gets called, and the component gets initialized properly:

```
        super(props)
```

5. Now, we can set the initial state by setting `this.state`. Initially, there will be no todo items, and the `filter` value will be set to `'all'`:

```
        this.state = { todos: [], filteredTodos: [], filter: 'all'
}
    }
```

6. Then, we define the `componentDidMount` life cycle method, which is going to fetch todo items when the component first renders:

```
    componentDidMount () {
         this.fetchTodos()      }
```

7. Now, we are going to define the actual `fetchTodos` method, which in our case, is simply going to set the state, because we are not going to connect this simple app to a backend. We are also going to call `this.filterTodos()` in order to update the `filteredTodos` array after fetching todos:

```
    fetchTodos () {
        fetchAPITodos().then((todos) => {
            this.setState({ todos })
            this.filterTodos()
        })
    }
```

8. Next, we define the `addTodo` method, which creates a new item, and adds it to the state array, similar to what we did in our blog app using Hooks:

```
addTodo (title) {
    const { todos } = this.state

    const newTodo = { id: generateID(), title, completed: false }

    this.setState({ todos: [ newTodo, ...todos ] })
    this.filterTodos()     }
```

9. Then, we define the `toggleTodo` method, which uses the `map` function to find and modify a certain todo item:

```
toggleTodo (id) {
    const { todos } = this.state

    const newTodos = todos.map(t => {
        if (t.id === id) {
            return { ...t, completed: !t.completed }
        }

        return t
    }, [])

    this.setState({ todos: newTodos })
    this.filterTodos()     }
```

10. Now, we define the `removeTodo` method, which uses the `filter` function to find and remove a certain todo item:

```
removeTodo (id) {
    const { todos } = this.state
    const newTodos = todos.filter(t => {
        if (t.id === id) {
            return false
        }
         return true
    })

    this.setState({ todos: newTodos })
    this.filterTodos()     }
```

11. Then, we define a method to apply a certain `filter` to our todo items:

```
applyFilter (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
    }
}
```

12. Now, we can define the `filterTodos` method, which is going to call the `applyFilter` method, and update the `filteredTodos` array and the `filter` value:

```
filterTodos (filterArg) {
    this.setState(({ todos, filter }) => ({
        filter: filterArg || filter,
        filteredTodos: this.applyFilter(todos, filterArg || filter)
    }))
}
```

We are using `filterTodos` in order to re-filter todos after dding/removing items, as well as changing the filter. To allow both unctionalities to work correctly, we need to check whether the `filter` rgument, `filterArg`, was passed. If not, we fall back to the current `ilter` argument from the `state`.

13. Then, we adjust the `render` method in order to use state to provide a value for the `StateContext`, and we pass certain methods to the components:

```
render () {
    const { filter, filteredTodos } = this.state
    return (
        <StateContext.Provider value={filteredTodos}>
            <div style={{ width: 400 }}>
                <Header />
                <AddTodo addTodo={this.addTodo} />
                <hr />
                <TodoList toggleTodo={this.toggleTodo}
removeTodo={this.removeTodo} />
                <hr />
                <TodoFilter filter={filter}
filterTodos={this.filterTodos} />
            </div>
        </StateContext.Provider>
    )
}
```

14. Finally, we need to re-bind `this` to the class, so that we can pass the methods to our components without the `this` context changing. Adjust the `constructor` as follows:

```
constructor () {
    super(props)
    this.state = { todos: [], filteredTodos: [], filter: 'all' }
    this.fetchTodos = this.fetchTodos.bind(this)
    this.addTodo = this.addTodo.bind(this)
    this.toggleTodo = this.toggleTodo.bind(this)
    this.removeTodo = this.removeTodo.bind(this)
    this.filterTodos = this.filterTodos.bind(this)
}
```

Now, our `App` component can dynamically fetch, add, toggle, remove, and filter todo items. As we can see, when we use class components, we need to re-bind the `this` context of the handler functions to the class.

Step 4: Making the AddTodo component dynamic

After making our `App` component dynamic, it is time to make all of our other components dynamic as well. We are going to start from the top, with the `AddTodo` component.

Let's make the `AddTodo` component dynamic now:

1. In `src/AddTodo.js`, we first define a `constructor`, which sets the initial `state` for the `input` field:

```
export default class AddTodo extends React.Component {
    constructor (props) {
        super(props)
        this.state = {
            input: ''
        }
    }
```

2. Then, we define a method for handling changes in the `input` field:

```
    handleInput (e) {
        this.setState({ input: e.target.value })     }
```

3. Now, we are going to define a method that can handle a new todo item being added:

```
    handleAdd () {
        const { input } = this.state
        const { addTodo } = this.props
        if (input) {
            addTodo(input)
            this.setState({ input: '' })
        }
    }
```

4. Next, we can assign the state value and handler methods to the `input` field and button:

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

5. Finally, we need to adjust the `constructor` in order to re-bind the `this` context for all of the handler methods:

```
constructor () {
    super(props)

    this.state = {
        input: ''
    }

    this.handleInput = this.handleInput.bind(this)

    this.handleAdd = this.handleAdd.bind(this)

}
```

Now, our `AddTodo` component will show a disabled button as long as no text is entered. When activated, clicking the button will trigger the `handleAdd` function that has been passed down from the `App` component.

Step 5: Making the TodoList component dynamic

The next component in our ToDo app is the `TodoList` component. Here, we just need to get the todo items from the `StateContext`.

Let's make the `TodoList` component dynamic now:

1. In `src/TodoList.js`, we first import the `StateContext`, below the `TodoItem` import statement:

```
import StateContext from './StateContext'
```

2. Then, we set the `contextType` to the `StateContext`, which will allow us to access the context via `this.context`:

```
export default class TodoList extends React.Component {
    static contextType = StateContext
```

With class components, if we want to use multiple contexts, we have to se the `StateContext.Consumer` component, as follows:

```
<StateContext.Consumer>{value => <div>State is:
{value}</div>}</StateContext.Consumer>.
```

As you can imagine, using multiple contexts like this, will result in a very eep component tree (wrapper hell), and our code will be hard to read and refactor.

3. Now, we can get the items from `this.context` instead of statically defining them:

```
render () {
    const items = this.context
```

4. Finally, we pass all props to the `TodoItem` component so that we can use the `removeTodo` and `toggleTodo` methods there:

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

Now, our `TodoList` component gets the items from the `StateContext` instead of statically defining them.

Step 6: Making the TodoItem component dynamic

Now that we have passed on the `removeTodo` and `toggleTodo` methods as props to the `TodoItem` component, we can implement these features there.

Let's make the `TodoItem` component dynamic now:

1. In `src/TodoItem.js`, we start by defining the handler methods for the `toggleTodo` and `removeTodo` functions:

```
handleToggle () {
    const { toggleTodo, id } = this.props
    toggleTodo(id)
}

handleRemove () {
    const { removeTodo, id } = this.props
    removeTodo(id)     }
```

2. Then, we assign the handler methods to the `checkbox` and `button`, respectively:

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

3. Finally, we need to re-bind the `this` context for the handler methods. Create a new `constructor`, as follows:

```
export default class TodoItem extends React.Component {
    constructor (props) {
        super(props)

        this.handleToggle = this.handleToggle.bind(this)
        this.handleRemove = this.handleRemove.bind(this)
    }
```

Now, the `TodoItem` component triggers the toggle and remove handler functions.

Step 7: Making the TodoFilter component dynamic

Lastly, we are going to use the `filterTodos` method to dynamically filter our todo item list.

Let's start making the `TodoFilter` component dynamic:

1. In `src/TodoFilter.js`, in the `TodoFilter` class, we pass all props down to the `TodoFilterItem` components:

```
export default class TodoFilter extends React.Component {
    render () {
        return (
            <div>
                <TodoFilterItem {...this.props} name="all" />{' /
'}
                <TodoFilterItem {...this.props} name="active" />{'
/ '}
                <TodoFilterItem {...this.props} name="completed" />
            </div>
        )
    }
}
```

2. In `src/TodoFilter.js`, in the `TodoFilterItem` class, we first define a handler method for setting the filter:

```
handleFilter () {
    const { name, filterTodos } = this.props
    filterTodos(name)     }
```

3. We then get the `filter` prop from `TodoFilter`:

```
render () {
    const { name, filter = 'all' } = this.props
```

4. Next, we use the `filter` prop to display the currently selected filter in `bold`:

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

5. Then, we bind the handler method—via `onClick`—to the filter item:

```
        return <span style={style}
 onClick={this.handleFilter}>{name}</span>
    }
```

6. Finally, we create a new `constructor` for the `TodoFilterItem` class, and rebind the `this` context of the handler method:

```
class TodoFilterItem extends React.Component {
    constructor (props) {
        super(props)
        this.handleFilter = this.handleFilter.bind(this)

    }
```

Now, our `TodoFilter` component triggers the `handleFilter` method in order to change the filter. Our whole app is dynamic now, and we can use all of its functionalities.

Exercise 4: Migrating from React class components (Chapter11_2)

After setting up our example project with React class components, we are now going to migrate this project to React Hooks. We are going to show how to migrate side effects, such as fetching todos when the component mounts, as well as state management, which we used for the inputs.

In this section, we are going to migrate the following components:

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

Step 1: Migrating the TodoItem component

One of the simplest components to migrate is the `TodoItem` component. It does not use any state or side effects so we can simply convert it to a function component.

Let's start migrating the `TodoItem` component:

1. Edit `src/TodoItem.js` and remove the class component code. We are going to define a function component instead now.
2. We start by defining the function, which accepts five props—the `title` value, the `completed` boolean, the `id` value, the `toggleTodo` function, and the `removeTodo` function:

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

3. Next, we define our two handler functions:

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

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

4. Finally, we return JSX code in order to render our component:

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

Try to keep your function components small, and combine them by creating new function components that wrap them. It is always a good idea to have many small components, rather than one large component. They are much easier to maintain, reuse, and refactor.

As we can see, function components do not require us to re-bind `this`, or to define constructors at all. Furthermore, we do not need to destructure from `this.props` multiple times. We can simply define all props in the header of our function.

Step 2: Migrating the TodoList component

Next, we are going to migrate the `TodoList` component, which wraps the `TodoItem` component. Here, we use a context, which means that we can now use a Context Hook.

Let's migrate the `TodoList` component now:

1. Edit `src/TodoList.js` and import the `useContext` Hook from React:

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

2. Remove the class component code. We are going to define a function component instead now.
3. We start by defining the header of our function. In this case, we do not destructure props, but simply store them in a `props` object:

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

4. Now we define the Context Hook:

```
    const items = useContext(StateContext)
```

5. Finally, we return the list of rendered `items`, passing the `item` and `props` objects to it using destructuring:

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

We define the `key` prop last, in order to avoid overwriting it with the destructuring of the `item` and `props` objects.

As we can see, using contexts with Hooks is much more straightforward. We can simply call a function, and use the return value. No magical assignment of `this.context` or wrapper hell when using multiple contexts!

Furthermore, we can see that we can gradually migrate components to React Hooks, and our app will still work. There is no need to migrate all components to Hooks at once. React class components can work well together with function React components that use Hooks. The only limitation is that we cannot use Hooks in class components. Therefore, we need to migrate a whole component at a time.

Step 3: Migrating the TodoFilter component

Next up is the `TodoFilter` component, which is not going to use any Hooks. However, we are going to replace the `TodoFilterItem` and `TodoFilter` components with two function components: one for the `TodoFilterItem`, and one for the `TodoFilter` component.

Migrating TodoFilterItem

First of all, we are going to migrate the `TodoFilterItem` component. Let's start migrating the component now:

1. Edit `src/TodoFilter.js` and remove the class component code. We are going to define a function component instead now.

2. Define a function for the `TodoFilterItem` component, which is going to accept three props—the `name` value, the `filterTodos` function, and the `filter` value:

```
function TodoFilterItem ({ name, filterTodos, filter = 'all' }) {
```

3. In this function, we define a handler function for changing the filter:

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

4. Next, we define a `style` object for our `span` element:

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

5. Finally, we return and render the `span` element:

```
return <span style={style} onClick={handleFilter}>{name}</span> }
```

As we can see, a function component requires much less boilerplate code than the corresponding class component.

Migrating TodoFilter

Now that we have migrated the `TodoFilterItem` component, we can migrate the `TodoFilter` component. Let's migrate it now:

1. Edit `src/TodoFilter.js` and remove the class component code. We are going to define a function component instead now.
2. Define a function for the `TodoFilter` component. We are not going to use destructuring on the props here:

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

3. In this component, we only return and render three `TodoFilterItem` components—passing the `props` down to them:

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

Now, our `TodoFilter` component has been successfully migrated.

Step 4: Migrating the AddTodo component

Next, we are going to migrate the `AddTodo` component. Here, we are going to use a State Hook to handle the `input` field state.

Let's migrate the `AddTodo` component now:

1. Edit `src/AddTodo.js` and adjust the import statement to import the `useState` Hook from React:

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

2. Remove the class component code. We are going to define a function component instead now.
3. First, we define the function, which accepts only one prop—the `addTodo` function:

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

4. Next, we define a State Hook for the `input` field state:

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

5. Now we can define the handler functions for the `input` field and the **add** button:

```
function handleInput (e) {
    setInput(e.target.value)     }

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

6. Finally, we return and render the `input` field and the **add** button:

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

As we can see, using the State Hook makes state management much simpler. We can define a separate value and setter function for each state value, instead of having to deal with a state object. Furthermore, we do not need to destructure from `this.state` all the time. As a result, our code is much more clean and concise.

Exercise 5: Migrating the App component

Lastly, all that is left to do is migrating the `App` component. Then, our whole ToDo app will have been migrated to React Hooks. Here, we are going to use a Reducer Hook to manage the state, an Effect Hook to fetch todos when the component mounts, and a Memo Hook to store the filtered todos list.

In this section, we are going to do the following:

- Define the actions
- Define the reducers
- Migrate the `App` component

Defining the actions

Our app is going to accept 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' }`

After defining the actions, we can move on to defining the reducers.

Step 1: Defining the reducers

We are now going to define the reducers for our state. We are going to need one app reducer and two sub-reducers: one for the todos and one for the filter.

The filtered todos list is going to be computed on the fly by the `App` component. We can later use a Memo Hook to cache the result and avoid unnecessary re-computation of the filtered todos list.

Defining the filter reducer

We are going to start by defining the reducer for the `filter` value. Let's define the filter reducer now:

1. Create a new `src/reducers.js` file and import the `generateID` function from the `src/api.js` file:

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

2. In the `src/reducers.js` file, define a new function, which is going to handle the `FILTER_TODOS` action, and set the value accordingly:

```
function filterReducer (state, action) {
    if (action.type === 'FILTER_TODOS') {
        return action.filter
    } else {
        return state
    }
}
```

Now, the `filterReducer` function is defined, and we can handle the `FILTER_TODOS` action properly.

Defining the todos reducer

Next, we are going to define a function for the todo items. Here, we are going to handle the `FETCH_TODOS`, `ADD_TODO`, `TOGGLE_TODO` and `REMOVE_TODO` actions.

Let's define the `todosReducer` function now:

1. In the `src/reducers.js` file, define a new function, which is going to handle these actions:

```
function todosReducer (state, action) {
    switch (action.type) {
```

2. For the `FETCH_TODOS` action, we simply replace the current state with the new `todos` array:

```
        case 'FETCH_TODOS':
            return action.todos
```

3. For the `ADD_TODO` action, we are going to insert a new item at the beginning of the current state array:

```
        case 'ADD_TODO':
            const newTodo = {
                id: generateID(),
                title: action.title,
                completed: false
            }
            return [ newTodo, ...state ]
```

4. For the `TOGGLE_TODO` action, we are going to use the `map` function to update a single todo item:

```
        case 'TOGGLE_TODO':
            return state.map(t => {
                if (t.id === action.id) {
                    return { ...t, completed: !t.completed }
                }
                return t
            }, [])
```

5. For the `REMOVE_TODO` action, we are going to use the `filter` function to remove a single todo item:

```
        case 'REMOVE_TODO':
            return state.filter(t => {
                if (t.id === action.id) {
                    return false
                }
                return true
            })
```

6. By default (for all other actions), we simply return the current `state`:

```
        default:
            return state
    }
}
```

Now, the todos reducer is defined, and we can handle the `FETCH_TODOS`, `ADD_TODO`, `TOGGLE_TODO` and `REMOVE_TODO` actions.

Defining the app reducer

Finally, we need to combine our other reducers into a single reducer for our app state. Let's define the `appReducer` function now:

1. In the `src/reducers.js` file, define a new function for the `appReducer`:

```
export default function appReducer (state, action) {
```

2. In this function, we return an object with the values from the other reducers. We simply pass the sub-state and action down to the other reducers:

```
    return {
        todos: todosReducer(state.todos, action),
        filter: filterReducer(state.filter, action)
    }
}
```

Now, our reducers are grouped together. So, we only have one `state` object and one `dispatch` function.

Step 2: Migrating the component

Now that we have defined our reducers, we can start migrating the `App` component. Let's migrate it now:

1. Edit `src/App.js` and adjust the import statement to import `useReducer`, `useEffect`, and `useMemo` from `React`:

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

2. Import the `appReducer` function from `src/reducers.js`:

```
import appReducer from './reducers'
```

3. Remove the class component code. We are going to define a function component instead now.
4. First, we define the function, which is not going to accept any props:

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

5. Now, we define a Reducer Hook using the `appReducer` function:

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

6. Next, we define an Effect Hook, which is going to fetch `todos` via the API function, and then a `FETCH_TODOS` action will be dispatched:

```
    useEffect(() => {
        fetchAPITodos().then((todos) =>
            dispatch({ type: 'FETCH_TODOS', todos })
        )
    }, [])
```

7. Then, we implement the filter mechanism using a Memo Hook, in order to optimize performance and avoid re-computing the filtered todos list when nothing changes:

```
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 ])
```

8. Now, we define various functions that are going to dispatch actions and change the state:

```
function addTodo (title) {
    dispatch({ type: 'ADD_TODO', title })
}

function toggleTodo (id) {
    dispatch({ type: 'TOGGLE_TODO', id })

}

function removeTodo (id) {

    dispatch({ type: 'REMOVE_TODO', id })
}

function filterTodos (filter) {
    dispatch({ type: 'FILTER_TODOS', filter })

}
```

9. Finally, we return and render all the components that are needed for our ToDo app:

```
    return (
        <StateContext.Provider value={filteredTodos}>
            <div style={{ width: 400 }}>
                <Header />
                <AddTodo addTodo={addTodo} />
                <hr />
                <TodoList toggleTodo={toggleTodo}
removeTodo={removeTodo} />
                <hr />
                <TodoFilter filter={state.filter}
filterTodos={filterTodos} />
            </div>
        </StateContext.Provider>
    )
}
```

As we can see, using a reducer to handle complex state changes makes our code much more concise and easier to maintain. Our app is now fully migrated to Hooks!