# 4: Using the Reducer and Effect Hooks

## Exercise 1: Reducer Hooks versus State Hooks

In the previous lab, we learned about dealing with local and global states. We used State Hooks for both cases, which is fine for simple state changes. However, when our state logic becomes more complicated, we are going to need to ensure that we keep the state consistent. In order to do so, we should use a Reducer Hook instead of multiple State Hooks, because it is harder to maintain synchronicity between multiple State Hooks that depend on each other. As an alternative, we could keep all state in one State Hook, but then we have to make sure that we do not accidentally overwrite parts of our state.

### Step 1: Problems with the State Hook

The State Hook already supports passing complex objects and arrays to it, and it can handle their state changes perfectly well. However, we are always going to have to change the state directly, which means that we need to use a lot of spread syntax, in order to make sure that we are not overwriting other parts of the state. For example, imagine that we have a state object like this:

```
const [ config, setConfig ] = useState({ filter: 'all', expandPosts: true })
```

Now, we want to change the filter:

```
setConfig({ filter: { byAuthor: 'Fname Lname', fromDate: '2019-04-29' } })
```

If we simply ran the preceding code, we would be removing the `expandPosts` part of our state! So, we need to do the following:

```
setConfig({ ...config, filter: { byAuthor: 'Fname Lname', fromDate:
'2019-04-29' } })
```

Now, if we wanted to change the `fromDate` filter to a different date, we would need to use spread syntax twice, to avoid removing the `byAuthor` filter:

```
setConfig({ ...config, filter: { ...config.filter, fromDate: '2019-04-30' } })
```

But, what happens if we do this when the `filter` state is still a string? We are going to get the following result:

```
{ filter: { '0': 'a', '1': 'l', '2': 'l', fromDate: '2019-04-30' },   expandPosts:
true }
```

What? Why are there suddenly three new keys—`0`, `1`, and `2`? This is because spread syntax also works on strings, which are spread in such a way that each letter gets a key, based on its index in the string.

As you can imagine, using spread syntax and changing the state object directly can become very tedious for larger state objects. Furthermore, we always need to make sure that we do not introduce any bugs, and we need to check for bugs in multiple places all across our app.

## Step 2: Actions

Instead of changing the state directly, we could make a function that deals with state changes. Such a function would only allow state changes via certain actions, such as a `CHANGE_FILTER` or a `TOGGLE_EXPAND` action.

Actions are simply objects that have a `type` key, telling us which action we are dealing with, and additional keys more closely describing the action.

The `TOGGLE_EXPAND` action is quite simple. It is simply an object with the action `type` defined:

```
{ type: 'TOGGLE_EXPAND' }
```

The `CHANGE_FILTER` action could deal with the complex state changes that we had problems with earlier, as follows:

```
{ type: 'CHANGE_FILTER', all: true }
{ type: 'CHANGE_FILTER', fromDate: '2019-04-29' }
{ type: 'CHANGE_FILTER', byAuthor: 'Fname Lname' }
{ type: 'CHANGE_FILTER', fromDate: '2019-04-30' }
```

The second, third, and fourth actions would change the `filter` state from a string to an object, and then set the respective key. If the object already exists, we would simply adjust the keys that were defined in the action. After each action, the state would change as follows:

```
{ expandPosts: true, filter: 'all' }
{ expandPosts: true, filter: { fromDate: '2019-04-29' } }
{ expandPosts: true, filter: { fromDate: '2019-04-29', byAuthor: 'Fname Lname' } }
{ expandPosts: true, filter: { fromDate: '2019-04-30', byAuthor: 'Fname Lname' } }
```

Now, take a look at the following code:

```
{ type: 'CHANGE_FILTER', all: true }
```

If we dispatched another action, as in the preceding code, then the state would go back to being the `all` string, as it was in the initial state.

## Step 3: Reducers

Now, we still need to define the function that handles these state changes. Such a function is known as a reducer function. It takes the current `state` and `action` as arguments, and returns a new state.

> If you are aware of the Redux library, you will already be very familiar with the concept of state, actions, and reducers.

Now, we are going to define our `reducer` function:

1. We start with the function definition of our `reducer`:

   ```
   function reducer (state, action) {
   ```

2. Then, we check for `action.type` using a `switch` statement:

   ```
       switch (action.type) {
   ```

3. Now, we are going to handle the `TOGGLE_EXPAND` action, where we simply toggle the current `expandPosts` state:

   ```
           case 'TOGGLE_EXPAND':
               return { ...state, expandPosts: !state.expandPosts }
   ```

4. Next, we are going to handle the `CHANGE_FILTER` action. Here, we first need to check if `all` is set to `true`, and, in that case, simply set our `filter` to the `'all'` string:

   ```
           case 'CHANGE_FILTER':
               if (action.all) {
                   return { ...state, filter: 'all' }
               }
   ```

5. Now, we have to handle the other `filter` options. First, we check if the `filter` variable is already an `object`. If not, we create a new one. Otherwise, we use the existing object:

   ```
               let filter = typeof state.filter === 'object' ?
                   state.filter : {}
   ```

6. Then, we define the handlers for the various filters, allowing for multiple filters to be set at once, by not immediately returning the new `state`:

   ```
               if (action.fromDate) {
                   filter = { ...filter, fromDate: action.fromDate }
               }
               if (action.byAuthor) {
                   filter = { ...filter, byAuthor: action.byAuthor }
               }
   ```

7. Finally, we return the new `state`:

   ```
               return { ...state, filter }
   ```

8. For the `default` case, we throw an error, because this is an unknown action:

```
            default:
                throw new Error()
        }
    }
```

Throwing an error in the default case is different to what is best practice with Redux reducers, where we would simply return the current state in the default case. Because React Reducer Hooks do not store all state in one object, we are only going to handle certain actions for certain state objects, so we can throw an error for unknown actions.

Now, our `reducer` function has been defined, and we can move on to defining the Reducer Hook.

## Step 4: The Reducer Hook

Now that we have defined actions and the `reducer` function, we can create a Reducer Hook from the `reducer`. The signature for the `useReducer` Hook is as follows:

```
const [ state, dispatch ] = useReducer(reducer, initialState)
```

The only thing that we still need to define is the `initialState`; then we can define a Reducer Hook:

```
const initialState = { all: true }
```

Now, we can access the state by using the `state` object that was returned from the Reducer Hook, and dispatch actions via the `dispatch` function, as follows:

```
dispatch({ type: 'TOGGLE_EXPAND' })
```

If we want to add additional options to the action, we simply add them to the action object:

```
dispatch({ type: 'CHANGE_FILTER', fromDate: '2019-04-30' })
```

As we can see, dealing with state changes using actions and reducers is much easier than having to adjust the state object directly.

## Implementing Reducer Hooks

After learning about actions, reducers, and the Reducer Hook, we are going to implement them in our blog app. Any existing State Hook can be turned into a Reducer Hook, when the state object or state changes become too complex.

> If there are multiple `setState` functions that are always called at the same time, it is a good hint that they should be grouped together in a single Reducer Hook.

Global state is usually a good candidate for using a Reducer Hook, rather than a State Hook, because global-state changes can happen anywhere in the app. Then, it is much easier to deal with actions, and update the state-changing logic only in one place. Having all the state-changing logic in one place makes it easier to maintain and fix bugs, without introducing new ones by forgetting to update the logic everywhere.

We are now going to turn some of the existing State Hooks in our blog app into Reducer Hooks.

## Turning a State Hook into a Reducer Hook

In our blog app, we have two global State Hooks, which we are going to replace with Reducer Hooks:

- `user` state
- `posts` state

We start by replacing the `user` State Hook.

## Exercise 2: Replacing the user State Hook

We are going to start with the `user` State Hook, because it is simpler than the `posts` State Hook. Later on, the `user` state will contain complex state changes, so it makes sense to use a Reducer Hook here.

First, we are going to define our actions, then we are going to define the reducer function.
Finally, we are going to replace the State Hook with a Reducer Hook.

### Step 1: Defining actions

We start by defining our actions, as these will be important when defining the reducer function.

Let's define the actions now:

1. First, we are going to need an action to allow a user to log in, by providing a `username` value and a `password` value:

    ```
    { type: 'LOGIN', username: 'Fname Lname', password: 'notsosecure' }
    ```

2. Then, we are also going to need a `REGISTER` action, which, in our case, is going to be similar to the `LOGIN` action, because we did not implement any registration logic yet:

    ```
    { type: 'REGISTER', username: 'Fname Lname', password:
    'notsosecure', passwordRepeat: 'notsosecure' }
    ```

3. Finally, we are going to need a `LOGOUT` action, which is simply going to log out the currently logged-in user:

    ```
    { type: 'LOGOUT' }
    ```

Now, we have defined all the required user-related actions and we can move on to defining the reducer function.

## Step 2: Defining the reducer

Next, we define a reducer function for the `user` state. For now, we are going to place our reducers in the `src/App.js` file.

> Later on, it might make sense to create a separate `src/reducers.js` file, or even a separate `src/reducers/` directory, with separate files for each reducer function.

Let's start defining the `userReducer` function:

1. In the `src/App.js` file, before the `App` function definition, create a `userReducer` function for the `user` state:

   ```
   function userReducer (state, action) {
   ```

2. Again, we use a `switch` statement for the `action` type:

   ```
       switch (action.type) {
   ```

3. Then, we handle the `LOGIN` and `REGISTER` actions, where we set the `user` state to the given `username` value. In our case, we simply return the `username` value from the `action` object for now:

   ```
           case 'LOGIN':
           case 'REGISTER':
               return action.username
   ```

4. Next, we handle the `LOGOUT` action, where we set the state to an empty string:

   ```
           case 'LOGOUT':
               return ''
   ```

5. Finally, we throw an error when we encounter an unhandled action:

   ```
           default:
               throw new Error()
       }
   }
   ```

Now, the `userReducer` function is defined, and we can move on to defining the Reducer Hook.

## Step 3: Defining the Reducer Hook

After defining the actions and the reducer function, we are going to define the Reducer Hook, and pass its state and the dispatch function to the components that need it.

Let's start implementing the Reducer Hook:

1. First, we have to import the `useReducer` Hook, by adjusting the following `import` statement in `src/App.js`:

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

2. Edit `src/App.js`, and remove the following State Hook:

   ```
   const [ user, setUser ] = useState('')
   ```

   Replace the preceding State Hook with a Reducer Hook—the initial state is an empty string, as we had it before:

   ```
   const [ user, dispatchUser ] = useReducer(userReducer, '')
   ```

3. Now, pass the `user` state and the `dispatchUser` function to the `UserBar` component, as a `dispatch` prop:

   ```
   <UserBar user={user} dispatch={dispatchUser} />
   ```

4. We do not need to modify the `CreatePost` component, as we are only passing the `user` state to it, and that part did not change.
5. Next, we edit the `UserBar` component in `src/user/UserBar.js`, and replace the `setUser` prop with the `dispatch` function:

   ```
   export default function UserBar ({ user, dispatch }) {
       if (user) {
           return <Logout user={user} dispatch={dispatch} />
       } else {
           return (
               <React.Fragment>
                   <Login dispatch={dispatch} />
                   <Register dispatch={dispatch} />
               </React.Fragment>
           )
       }
   }
   ```

6. Now, we can edit the `Login` component in `src/user/Login.js`, and replace the `setUser` function with the `dispatch` function:

   ```
   export default function Login ({ dispatch }) {
   ```

7. Then, we replace the call to `setUser` with a call to the `dispatch` function, dispatching a `LOGIN` action:

   ```
               <form onSubmit={e => { e.preventDefault(); dispatch({ type: 'LOGIN', 
   username }) }}>
   ```

   We could also make functions that return actions—so-called action creators. Instead of manually creating the action object every time, we could simply call `loginAction('username')` instead, which returns the corresponding `LOGIN` action object.

8. We repeat the same process for the `Register` component in `src/user/Register.js`:

   ```
   export default function Register ({ dispatch }) {
       // ...
               <form onSubmit={e => { e.preventDefault(); dispatch({ type: 
   'REGISTER', username }) }}>
   ```

9. Finally, we also repeat the same process for the `Logout` component in `src/user/Logout.js`:

   ```
   export default function Logout ({ user, dispatch }) {
       // ...
               <form onSubmit={e => { e.preventDefault(); dispatch({ type: 'LOGOUT' 
   }) }}>
   ```

Now, our app should work the same way as before, but it uses the Reducer Hook instead of a simple State Hook!

## Exercise 3: Replacing the posts State Hook

It also makes sense to use a Reducer Hook for the `posts` state, because, later on, we are going to have features that can be used to delete and edit posts, so it makes a lot of sense to keep these complex state changes contained. Let's now get started replacing the posts State Hook with a Reducer Hook.

### Step 1: Defining actions

Again, we start by defining actions. At the moment, we are only going to consider a `CREATE_POST` action:

```
{ type: 'CREATE_POST', title: 'React Hooks', content: 'The greatest thing since sliced
bread!', author: 'Fname Lname' }
```

That is the only action we are going to need for posts, at the moment.

### Step 2: Defining the reducer

Next, we are going to define the reducer function in a similar way that we did for the `user` state:

1. We start by editing `src/App.js`, and defining the reducer function there. The following code defines the `postsReducer` function:

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

2. In this function, we are going to handle the `CREATE_POST` action. We first create a `newPost` object, and then we insert it at the beginning of the current `posts` state by using spread syntax, in a similar way to how we did it in the `src/post/CreatePost.js` component earlier:

```
        case 'CREATE_POST':
            const newPost = { title: action.title, content:
                action.content, author: action.author }
            return [ newPost, ...state ]
```

3. For now, this will be the only action that we handle in this reducer, so we can now define the `default` statement:

```
        default:
            throw new Error()
    }
}
```

Now, the `postsReducer` function is defined, and we can move on to creating the Reducer Hook.

### Step 3: Defining the Reducer Hook

Finally, we are going to define and use the Reducer Hook for the `posts` state:

1. We start by removing the following State Hook in `src/App.js`:

```
const [ posts, setPosts ] = useState(defaultPosts)
```

We replace it with the following Reducer Hook:

```
const [ posts, dispatchPosts ] = useReducer(postsReducer,
   defaultPosts)
```

2. Then, we pass the `dispatchPosts` function to the `CreatePost` component, as a `dispatch` prop:

```
            {user && <CreatePost user={user} posts={posts}
dispatch={dispatchPosts} />}
```

3. Next, we edit the `CreatePost` component in `src/post/CreatePost.js`, and replace the `setPosts` function with the `dispatch` function:

```
export default function CreatePost ({ user, posts, dispatch }) {
```

4. Finally, we use the `dispatch` function in the `handleCreate` function:

```
function handleCreate () {
    dispatch({ type: 'CREATE_POST', title, content, author: user })
}
```

Now, the `posts` state also uses a Reducer Hook instead of a State Hook, and it works in the same way as before! However, if we want to add more logic for managing posts, such as searching, filtering, deleting, and editing, later on, it will be much easier to do so.

## Exercise 4: Merging Reducer Hooks(chapter4_2)

Currently, we have two different dispatch functions: one for the `user` state, and one for the `posts` state. In our case, it makes sense to combine the two reducers into one, which then calls further reducers, in order to deal with the sub-state.

This pattern is similar to the way in which reducers work in Redux, where we only have one object containing the whole state tree of the application, which in the case of the global state, makes sense. However, for complex local state changes, it might make more sense to keep the reducers separate.

**Step 1:** Let's start merging our reducer functions into one reducer function. While we are at it, let's refactor all the reducers into a `src/reducers.js` file, in order to keep the `src/App.js` file more readable:

1. Create a new `src/reducers.js` file.

2. Cut the following code from the `src/App.js` file, and paste it into the `src/reducers.js` file:

```
function userReducer (state, action) {
    switch (action.type) {
         case 'LOGIN':
        case 'REGISTER':
            return action.username
        case 'LOGOUT':
            return ''
        default:
            throw new Error()
    }
}

function postsReducer (state, action) {
    switch (action.type) {
        case 'CREATE_POST':
            const newPost = { title: action.title, content:
action.content, author: action.author }
            return [ newPost, ...state ]
        default:
            throw new Error()
    }
}
```

3. Edit `src/reducers.js`, and define a new reducer function below the existing reducer functions, called `appReducer`:

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

4. In this `appReducer` function, we are going to call the other two reducer functions, and return the full state tree:

```
    return {
        user: userReducer(state.user, action),
        posts: postsReducer(state.posts, action)
    }
}
```

5. Edit `src/App.js`, and import the `appReducer` there:

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

6. Then, we remove the following two Reducer Hook definitions:

```
    const [ user, dispatchUser ] = useReducer(userReducer,
     '')
    const [ posts, dispatchPosts = useReducer(postsReducer,
 defaultPosts)
```

We replace the preceding Reducer Hook definitions with a single Reducer Hook definition for the `appReducer`:

```
    const [ state, dispatch ] = useReducer(appReducer, { user: '', posts:
defaultPosts })
```

7. Next, we extract the `user` and `posts` values from our `state` object, using destructuring:

```
            const { user, posts } = state
```

8. Now, we still need to replace the `dispatchUser` and `dispatchPosts` functions that we passed to the other components with the `dispatch` function:

```
                <UserBar user={user} dispatch={dispatch} />
                <br />
                {user && <CreatePost user={user} posts={posts}
    dispatch={dispatch} />}
```

As we can see, now there is only one `dispatch` function, and a single state object.

## Step 2: Ignoring unhandled actions

However, if we try logging in now, we are going to see an error from the `postsReducer`. This is because we are still throwing an error on unhandled actions. In order to avoid this, we have to instead ignore unhandled actions, and simply return the current state:

Edit the `userReducer` and `postsReducer` functions in `src/reducers.js`, and remove the following code:

```
        default:
            throw new Error()
```

Replace the preceding code with a `return` statement that returns the current `state`:

```
        default:
            return state
```

As we can see, now our app still works in exactly the same way as before, but we are using a single reducer for our whole app state!

## Exercise 5: Using Effect Hooks

The last essential Hook that we are going to be using frequently is the Effect Hook. Using the Effect Hook, we can perform side effects from our components, such as fetching data when the component mounts or updates.

In the case of our blog, we are going to implement a feature that updates the title of our web page when we log in, so that it contains the username of the currently logged-in user.

### Step 1: Remember componentDidMount and componentDidUpdate?

If you have worked with React before, you have probably used the `componentDidMount` and `componentDidUpdate` life cycle methods. For example, we can set the document `title` to a given prop as follows, using a React class component. In the following code section, the life cycle method is highlighted in bold:

```
import React from 'react'

class App extends React.Component {
     componentDidMount () {
        const { title } = this.props
        document.title = title
    }
    render () {
        return (
            <div>Test App</div>
        )
    }
}
```

This works fine. However, when the `title` prop updates, the change does not get reflected in the title of our web page. To solve this problem, we need to define the `componentDidUpdate` life cycle method (new code in bold), as follows:

```
import React from 'react'
class App extends React.Component {
     componentDidMount () {
        const { title } = this.props
        document.title = title
    }
    componentDidUpdate (prevProps) {
        const { title } = this.props
        if (title !== prevProps.title) {
            document.title = title
        }
    }
    render () {
        return (
            <div>Test App</div>
        )
    }
}
```

You might have noticed that we are writing almost the same code twice; therefore, we could create a new method to deal with updates to `title`, and then call it from both life cycle methods. In the following code section, the updated code is highlighted in bold:

```
import React from 'react'

class App extends React.Component {
    updateTitle () {
        const { title } = this.props
        document.title = title
    }
    componentDidMount () {
         this.updateTitle()
    }
    componentDidUpdate (prevProps) {
        if (this.props.title !== prevProps.title) {
            this.updateTitle()
        }
    }
    render () {
        return (
            <div>Test App</div>
        )
    }
}
```

However, we still need to call `this.updateTitle()` twice. When we update the code later on, and, for example, pass an argument to `this.updateTitle()`, we always need to remember to pass it in both calls to the method. If we forget to update one of the life cycle methods, we might introduce bugs. Furthermore, we need to add an `if` condition to `componentDidUpdate`, in order to avoid calling `this.updateTitle()` when the `title` prop did not change.

## Step 2: Using an Effect Hook

In the world of Hooks, the `componentDidMount` and `componentDidUpdate` life cycle methods are combined in the `useEffect` Hook, which—when not specifying a dependency array—triggers whenever any props in the component change.

So, instead of using a class component, we can now define a function component with an Effect Hook, which does the same thing as before. The function passed to the Effect Hook is called "effect function":

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

function App ({ title }) {
    useEffect(() => {
        document.title = title     })
    return (
        <div>Test App</div>
    )
}
```

And that's all we need to do! The Hook that we have defined will call our effect function every time any props change.

## Step 3: Trigger effect only when certain props change

If we want to make sure that our effect function only gets called when the `title` prop changes, for example, for performance reasons, we can specify which values should trigger the changes, as a second argument to the `useEffect` Hook:

```
useEffect(() => {
    document.title = title
}, [title])
```

And this is not just restricted to props, we can use any value here, even values from other Hooks, such as a State Hook or a Reducer Hook:

```
const [ title, setTitle ] = useState('')
useEffect(() => {
    document.title = title
}, [title])
```

As we can see, using an Effect Hook is much more straightforward than using life cycle methods when dealing with changing values.

## Step 4: Trigger effect only on mount

If we want to replicate the behavior of only adding a `componentDidMount` life cycle method, without triggering when the props change, we can do this by passing an empty array as the second argument to the `useEffect` Hook:

```
useEffect(() => {
    document.title = title
}, [])
```

Passing an empty array means that our effect function will only trigger once when the component mounts, and it will not trigger when props change. However, instead of thinking about the mounting of components, with Hooks, we should think about the dependencies of effects. In this case, the effect does not have any dependencies, which means it will only trigger once. If an effect has dependencies specified, it will trigger again when any of the dependencies change.

## Step 5: Cleaning up effects

Sometimes effects need to be cleaned up when the component unmounts. To do so, we can return a function from the effect function of the Effect Hook. This returned function works similarly to the `componentWillUnmount` life cycle method:

```
    useEffect(() => {
        const updateInterval = setInterval(() => console.log('fetching update'),
updateTime)
        return () => clearInterval(updateInterval)
    }, [updateTime])
```

The code marked in bold above is called the cleanup function. The cleanup function will be called when the component unmounts and before running the effect again. This avoids bugs when, for example, the `updateTime` prop changes. In that case, the previous effect will be cleaned up and a new interval with the updated `updateTime` value will be defined.

## Step 6: Implementing an Effect Hook in our blog app (chapter4_3)

Now that we have learned how the Effect Hook works, we are going to use it in our blog app, to implement the title changing when we log in/log out (when the `user` state changes).

Let's get started implementing an Effect Hook in our blog app: 1. Edit
`src/App.js`, and import the `useEffect` Hook:

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

2. After defining our `useReducer` Hook and the state destructuring, define a `useEffect` Hook that adjusts the `document.title` variable, based on the `username` value:

```
useEffect(() => {
```

3. If the user is logged in, we set the `document.title` to `<username> - React Hooks Blog`. We use template strings for this, which allow us to include variables, or JavaScript expressions, in a string via the `${ }` syntax. Template strings are defined using `` ` ``:

```
if (user) {
           document.title = `${user} - React Hooks Blog`
```

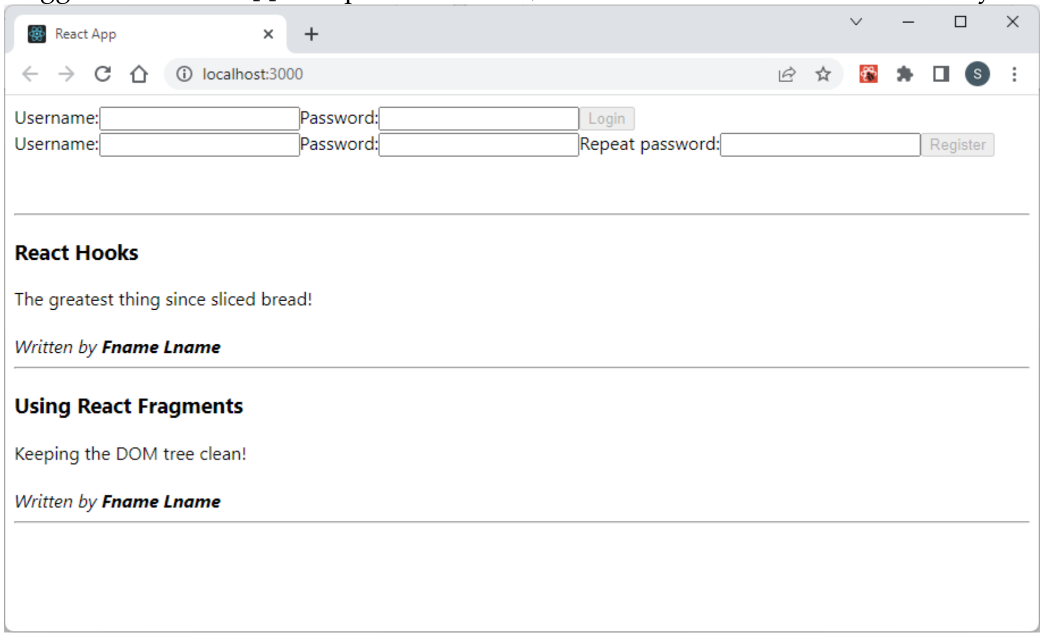
4. Otherwise, if the user is not logged in, we simply set the `document.title` to `React Hooks Blog`:

```
} else {
    document.title = 'React Hooks Blog'
}
```

5. Finally, we pass the `user` value as the second argument to the Effect Hook, in order to ensure that whenever the `user` value updates, our effect function triggers again:

```
}, [user])
```

If we start our app now, we can see that the `document.title` gets set to `React Hooks Blog`, because the Effect Hook triggers when the `App` component mounts, and the `user` value is not defined yet:



Username: Password:
Username: Password: Repeat password:

**React Hooks**

The greatest thing since sliced bread!

*Written by* ***Fname Lname***

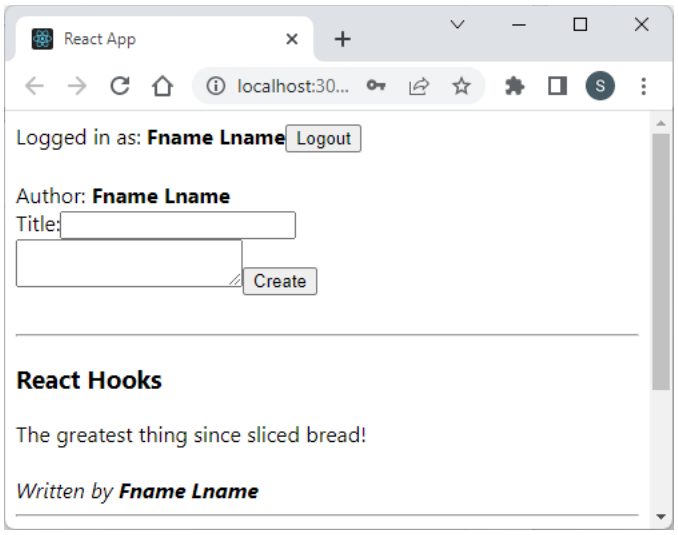
**Using React Fragments**

Keeping the DOM tree clean!

*Written by* ***Fname Lname***

The effect of our Effect Hook: changing the web page title

After logging in with `Test User`, we can see that the `document.title` changes to `Test User - React Hooks Blog`:



Logged in as: **Fname Lname**

Author: **Fname Lname**
Title:

**React Hooks**

The greatest thing since sliced bread!

*Written by* ***Fname Lname***

As we can see, our Effect Hook re-triggers successfully after the `user` value changes!