3 Writing Your First Application with React Hooks

Structuring React projects

After learning about the principles of React, how to use the `useState` Hook, and how Hooks work internally, we are now going to make use of the real `useState` Hook in order to develop a blog application. First, we are going to create a new project, and structure the folders in a way that will allow us to scale the project later on. Then, we are going to define the components that we are going to need in order to cover the basic features of a blog application. Finally, we are going to use Hooks to introduce state to our application! Throughout this lab, we are also going to learn about **JSX**, and new JavaScript features that have been introduced in **ES6**, up to **ES2018**.

Folder structure

There are many ways that projects can be structured, and different structures can do well for different projects. Usually, we create a `src/` folder, and group our files there by features. Another popular way to structure projects is to group them by routes. For some projects, it might make sense to additionally separate by the kind of code, such as `src/api/` and `src/components/`. However, for our project, we are mainly going to focus on the **user interface** (**UI**). As a result, we are going to group our files by features in the `src/` folder.

> It is a good idea to start with a simple structure at first, and only nest more deeply when you actually need it. Do not spend too much time thinking about the file structure when starting a project, because usually, you do not know up front how files should be grouped

Choosing the features

We first have to think about which features we are going to implement in our blog application. At the bare minimum, we want to implement the following features:

- Registering users
- Logging in/out
- Viewing a single post
- Creating a new post Listing posts

Now that we have chosen the features, let's come up with an initial folder structure.

Coming up with an initial structure

From our previous functionalities, we can abstract a couple of feature groups:

- User (registering, log in/log out)
- Post (creating, viewing, listing)

We could now just keep it very simple, and create all of the components in the `src/` folder, without any nesting. However, since we already have quite a clear picture on the features that a blog application is going to need, we can come up with a simple folder structure now:

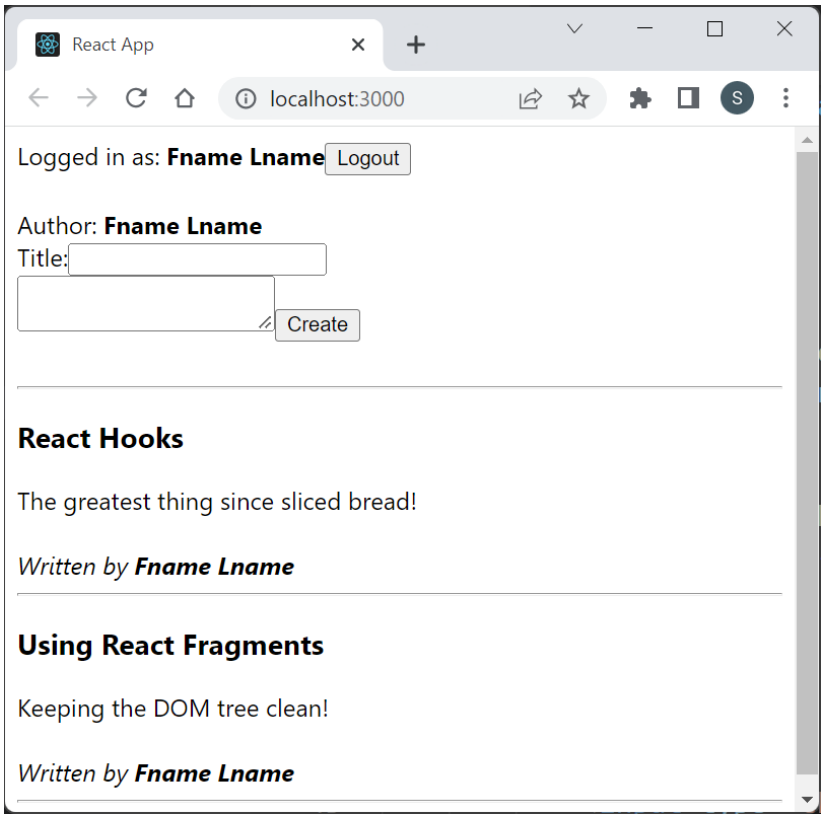
- `src/`
- `src/user/`
- `src/post/`

After defining the folder structure, we can move on to the component structure.

Component structure

The idea of components in React is to have each component deal with a single task or UI element. We should try to make components as fine-grained as possible, in order to be able to reuse code. If we find ourselves copying and pasting code from one component to another, it might be a good idea to create a new component, and reuse it in multiple other components.

Usually, when developing software, we start with a UI mock-up. For our blog application, a mock-up would look as follows:



Initial mock-up of our blog application

When splitting components, we use the single responsibility principle, which states that every module should have responsibility over a single encapsulated part of the functionality.

Exercise 1:

In this mock-up, we can draw boxes around each component and subcomponent, and give them names. Keep in mind that each component should have exactly one responsibility.

Step 1: We start with the fundamental components that make up this app:

Defining the fundamental components from our mock-up

We defined a `Logout` component for the logout feature, a `CreatePost` component, which contains the form to create a new post, and a `Post` component to display the actual posts.

Step 2: Now that we have defined our fundamental components, we are going to look at which components logically belong together, thereby forming a group. To do so, we now define the container components, which we need in order to group the components together:

Defining the container components from our mock-up

We defined a `PostList` component in order to group posts together, then a `UserBar` component in order to deal with login/logout and registration. Finally, we defined an `App` component in order to group everything together, and define the structure of our app.

Now that we are done with structuring our React project, we can move on to implementing the static components.

Exercise 2: Implementing static components (chapter3_2)

Before we start adding state via Hooks to our blog application, we are going to model the basic features of our application as static React components. Doing this means that we have to deal with the static view structure of our application.
It makes sense to deal with the static structure first, so as to avoid having to move dynamic code to different components later on. Furthermore, it is easier to deal only with **Hypertext Markup Language (HTML)** and CSS first—helping us to get started with projects quickly. Then, we can move on to implementing dynamic code and handling state.

Doing this step by step, instead of implementing everything at once, helps us to quickly get started with new projects without having to think about too much at once, and lets us avoid having to restructure projects later!

Step 1: Setting up the project

We have already learned how to set up a new React project. As we have learned, we can use the `create-react-app` tool to easily initialize a new project. We are going to do so now:

1. First, we use `create-react-app` to initialize our project:

   > **`npx create-react-app chapter3_1`**

2. Then, we create folders for our features:

   - **Create folder**: `src/user/`
   - **Create folder**: `src/post/`

Now that our project structure is set up, we can start implementing components.

Implementing users

We are going to start with the simplest feature in terms of static components: implementing user-related functionality. As we have seen from our mock-up, we are going to need four components here:

- A `Login` component, which we are going to show when the user is not logged in yet
- A `Register` component, which we are also going to show when the user is not logged in yet
- A `Logout` component, which is going to be shown after the user is logged in
- A `UserBar` component, which will display the other components conditionally

We are going to start by defining the first three components, which are all stand-alone components. Lastly, we will define the `UserBar` component, because it depends on the other components being defined.

Step 2: The Login component

First, we define the `Login` component, where we show two fields: a **Username** field, and a **Password** field. Furthermore, we show a **Login** button:

1. We start by creating a new file for our component: `src/user/Login.js`
2. In the newly created `src/user/Login.js` file, we import `React`:

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

3. Then, we define our function component. For now, the `Login` component will not accept any props:

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

4. Finally, we return the two fields and the **Login** button, via JSX. We also define a `form` container element to wrap them in. To avoid a page refresh when the form is submitted, we have to define an `onSubmit` handler and call `e.preventDefault()` on the event object:

```
    return (
        <form onSubmit={e => e.preventDefault()}>
            <label htmlFor="login-username">Username:</label>
            <input type="text" name="login-username"
               id="login-username" />
            <label htmlFor="login-password">Password:</label>
            <input type="password" name="login-password"
               id="login-password" />
            <input type="submit" value="Login" />
        </form>
    )
}
```

Here, we are using an anonymous function to define the `onSubmit` handler. Anonymous functions are defined as follows, if they do not have any arguments: `() => { ... }`, instead of `function () { ... }`. With arguments, we could write `(arg1, arg2) => { ... }`, instead of `function (arg1, arg2) { ... }`. We can omit the `()` brackets if we only have a single argument. Additionally, we can omit the `{}` brackets if we only have a single statement in our function, like this:

```
e => e.preventDefault().
```

Using semantic HTML elements such as `<form>` and `<label>` make your app easier to navigate for people using accessibility assistance software, such as screen readers. Furthermore, when using semantic HTML, keyboard shortcuts, such as submitting forms by pressing the return key, automatically work.

Our `Login` component is implemented, and is now ready to be tested.

Step 3: Testing out our component

Now that we have defined our first component, let's render it and see what it looks like:

1. First, we edit `src/App.js`, and remove all its contents.
2. Then, we start by importing `React` and the `Login` component:

```
import React from 'react'
import Login from './user/Login'
```

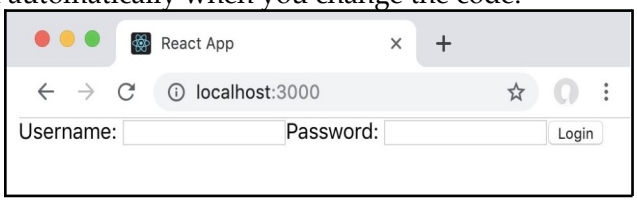
   It is a good idea to group imports in blocks of code that belong together. In this case, we separate external imports, such as React, from local imports, such as our `Login` component, by adding an empty line in between. Doing so keeps our code readable, especially when we add more import statements later.

3. Finally, we define the `App` component, and return the `Login` component:

```
export default function App () {
    return <Login />
}
```

   If we are only returning a single component, we can omit the brackets in the `return` statement. Instead of writing `return (<Login />)`, we can simply write `return <Login />`.

4. Open `http://localhost:3000` in your browser, and you should see the `Login` component being rendered. If you already had the page open in your browser, it should refresh automatically when you change the code:



The first component of our blog application: logging in by username and password

As we can see, the static `Login` component renders fine in React. We can now move on to the `Logout` component.

Step 4: The Logout component

Next, we define the `Logout` component, which is going to display the currently logged in user, and a button to log out:

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

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

3. This time, our function is going to take a `user` prop, which we are going to use to display the currently logged-in user:

```
export default function Logout ({ user }) {
```

Here we use destructuring in order to extract the `user` key from the `props` object. React passes all component props, in a single object, as the first argument to a function. Using destructuring on the first argument is similar to doing `const { user } = this.props` in a class component.

4. Finally, we return a text that shows the currently logged-in `user` and the **Logout** button:

```
    return (
        <form onSubmit={e => e.preventDefault()}>
            Logged in as: <b>{user}</b>
            <input type="submit" value="Logout" />
        </form>
    )
}
```

5. We can now replace the `Login` component with the `Logout` component in `src/App.js`, in order to see our newly defined component (do not forget to pass the `user` prop to it!):

```
import React from 'react'
import Logout from './user/Logout'

export default function App () {
    return <Logout user="Fname Lname" />
}
```

Now, the `Logout` component is defined, and we can move on to the `Register` component.

Step 5: The Register component

The static `Register` component will be very similar to the `Login` component, with an additional field to repeat the password. You might get the idea to merge them into one component if they are so similar, and add a prop to toggle the **Repeat password** field. However, it is best to stick to the single responsibility principle, and to have each component deal with only one functionality. Later on, we are going to extend the static components with dynamic code, and then `Register` and `Login` will have vastly different code. As a result, we would need to split them up again later.

Nevertheless, let's start working on the code for the `Register` component:

1. We start by creating a new `src/user/Register.js` file, and copying the code from the `Login` component, as the static components are very similar, after all. Make sure to change the name of the component to `Register`:

```
import React from 'react'

export default function Register () {
    return (
        <form onSubmit={e => e.preventDefault()}>
            <label htmlFor="register-username">Username:</label>
            <input type="text" name="register-username"
               id="register-username" />
            <label htmlFor="register-password">Password:</label>
            <input type="password" name="register-password"
               id="register-password" />
```

2. Next, we add the **Repeat password** field, right below the **Password** field code:

```
            <label htmlFor="register-password-repeat">Repeat
               password:</label>
            <input type="password" name="register-password-repeat"
  id="register-password-repeat" />
```

3. Finally, we also change the value of the submit button to **Register**:

```
            <input type="submit" value="Register" />
        </form>
    )
}
```

4. Again, we can edit `src/App.js` in order to show our component, in a similar way to how we did with the `Login` component:

```
import React from 'react'

import Register from './user/Register'


export default function App () {
    return <Register />
}
```

As we can see, our `Register` component looks very similar to the `Login` component.

Step 6: The UserBar component

Now it is time to put our user-related components together into a `UserBar` component. Here we are going to conditionally show either the `Login` and `Register` components, or the `Logout` component, depending on whether the user is already logged in or not.

Let's start implementing the `UserBar` component:

1. First, we create a new **`src/user/UserBar.js`** file, and import `React` as well as the three components that we defined:

```
import React from 'react'
import Login from './Login'
import Logout from './Logout'
import Register from './Register'
```

2. Next, we define our function component, and a value for the `user`. For now, we just save it in a static variable:

```
export default function UserBar () {

    const user = ''
```

3. Then, we check whether the user is logged in or not. If the user is logged in, we display the `Logout` component, and pass the `user` value to it:

```
    if (user) {
        return <Logout user={user} />
```

4. Otherwise, we show the `Login` and `Register` components. Here, we can use `React.Fragment` instead of a `<div>` container element. This keeps our UI tree clean, as the components will simply be rendered side by side, instead of being wrapped in another element:

```
    } else {
         return (
            <React.Fragment>
                <Login />
                <Register />
            </React.Fragment>
        )
    }
}
```

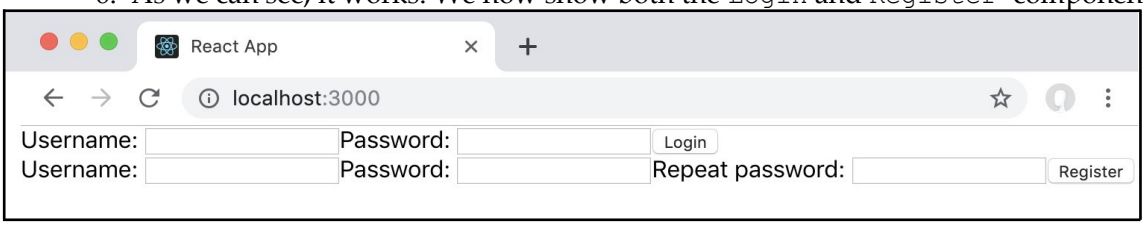
5. Again, we edit `src/App.js`, and now we show our `UserBar` component:

```
import React from 'react'

import UserBar from './user/UserBar'

export default function App () {
     return <UserBar />
}
```

6. As we can see, it works! We now show both the `Login` and `Register` components:

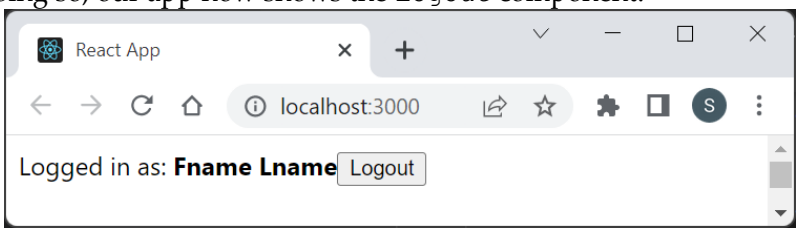


Our UserBar component, showing both the Login and Register components

7. Next, we can edit the `src/user/UserBar.js` file, and set the `user` value to a string:

```
const user = 'Daniel Bugl'
```

8. After doing so, our app now shows the `Logout` component:



Later on in this lab, we are going to add Hooks to our application, so that we can log in and have the state change dynamically without having to edit the code!

Exercise 3: Implementing posts

After implementing all the user-related components, we move on to implementing posts in our blog app. We are going to define the following components:

- A `Post` component to display a single post
- A `CreatePost` component for creating new posts
- A `PostList` component to show multiple posts

Let's get started implementing the post related components now.

Step 1: The Post component

We have already thought about which elements a post has when creating the mock-up. A post should have a title, content, and an author (the user who wrote the post).

Let's implement the `Post` component now:

1. First, we create a new file: `src/post/Post.js`
2. Then, we import `React`, and define our function component, accepting three props: `title`, `content`, and `author`:

```
import React from 'react'

export default function Post ({ title, content, author }) {
```

3. Next, we render all props in a way that resembles the mock-up:

```
    return (
        <div>
            <h3>{title}</h3>
            <div>{content}</div>
            <br />
            <i>Written by <b>{author}</b></i>
        </div>
    )
}
```

4. As always, we can test our component by editing the `src/App.js` file:

```
import React from 'react'

import Post from './post/Post'

export default function App () {
    return <Post title="React Hooks" content="The greatest thing
since sliced bread!" author="Daniel Bugl" />
}
```

Now, the static `Post` component has been implemented, and we can move on to the `CreatePost` component.

Step 2: The CreatePost component

Next, we implement a form to allow for the creation of new posts. Here, we pass the `user` value as a prop to the component, as the author should always be the currently logged-in user. Then, we show the author, and provide an input field for the `title`, and a `<textarea>` element for the content of the blog post.

Let's implement the `CreatePost` component now:

1. Create a new file: **`src/post/CreatePost.js`**
2. Define the following component:

```
import React from 'react'

export default function CreatePost ({ user }) {
    return (
        <form onSubmit={e => e.preventDefault()}>
            <div>Author: <b>{user}</b></div>
            <div>
                <label htmlFor="create-title">Title:</label>
                <input type="text" name="create-title"
                       id="create-title" />
            </div>
            <textarea />
            <input type="submit" value="Create" />
        </form>
    )
}
```

3. As always, we can test our component by editing the `src/App.js` file:

```
import React from 'react'

import CreatePost from './post/CreatePost'

export default function App () {
    return <CreatePost />
}
```

As we can see, the `CreatePost` component renders fine. We can now move on to the `PostList` component.

Step 3: The PostList component

After implementing the other post-related components, we can now implement the most important part of our blog app: the feed of blog posts. For now, the feed is simply going to show a list of blog posts.

Let's start implementing the `PostList` component now:

1. We start by importing `React` and the `Post` component:

```
import React from 'react'

import Post from './Post'
```

2. Then, we define our `PostList` function component, accepting a `posts` array as a prop. If `posts` is not defined, we set it to an empty array, by default:

```
export default function PostList ({ posts = [] }) {
```

3. Next, we render all `posts` by using the `.map` function and the spread syntax:

```
    return (
        <div>
            {posts.map((p, i) => <Post {...p} key={'post-' + i}
               />)}
        </div>
    )
}
```

If we are rendering a list of elements, we have to give each element a unique `key` prop. React uses this `key` prop to efficiently compute the difference of two lists, when the data has changed.

Here, we use the `map` function, which applies a function to all the elements of an array. This is similar to using a `for` loop, and storing all the results, but it is much more concise, declarative, and easier to read! Alternatively, we could do the following instead of using the `map` function:

```
let renderedPosts = []
let i = 0
for (let p of posts) {
    renderedPosts.push(<Post {...p} key={'post-' + i} />)
    i++
}

return (
    <div>
        {renderedPosts}
    </div>
)
```

We then return the `<Post>` component for each post, and pass all the keys from the post object, `p`, to the component as props. We do this by using the spread syntax, which has the same effect as listing all the keys from the object manually as props, as follows: `<Post title={p.title} content={p.content} author={p.author} />`

4. In the mock-up, we have a horizontal line after each blog post. We can implement this without an additional `<div>` container element, by using `React.Fragment`:

```
{posts.map((p, i) => (
     <React.Fragment key={'post-' + i} >
          <Post {...p} />
          <hr />
     </React.Fragment>
))}
```

The `key` prop always has to be added to the uppermost parent element that is rendered within the `map` function. In this case, we had to move the `key` prop from the `Post` component to the `React.Fragment` component.

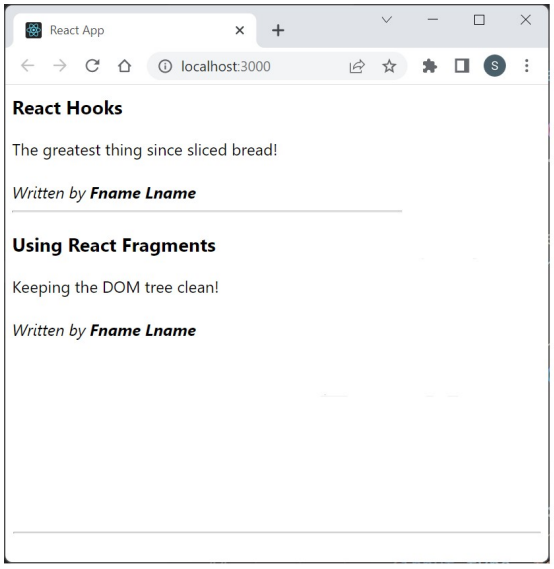
5. Again, we test our component by editing the `src/App.js` file:

```
import React from 'react'

import PostList from './post/PostList'

const posts = [
    { title: 'React Hooks', content: 'The greatest thing since
sliced bread!', author: 'Daniel Bugl' },
    { title: 'Using React Fragments', content: 'Keeping the DOM
tree clean!', author: 'Daniel Bugl' }
]
export default function App () {
     return <PostList posts={posts} />
}
```

Now, we can see that our app lists all the posts that we defined in the `posts` array:



As we can see, listing multiple posts via the `PostList` component works fine. We can now move on to putting the app together.

Step 4: Putting the app together

After implementing all components, in order to reproduce the mock-up, we now only have to put everything together in the `App` component. Then, we will have successfully reproduced the mock-up!

Let's start modifying the `App` component, and putting our app together:

1. Edit `src/App.js`, and remove all of the current code.
2. First, we import `React`, `PostList`, `CreatePost`, and the `UserBar` components:

```
import React from 'react'

import PostList from './post/PostList'
import CreatePost from './post/CreatePost'

import UserBar from './user/UserBar'
```

3. Then, we define some mock data for our app:

```
const user = 'Daniel Bugl'
const posts = [
    { title: 'React Hooks', content: 'The greatest thing since
sliced bread!', author: 'Daniel Bugl' },
    { title: 'Using React Fragments', content: 'Keeping the DOM
tree clean!', author: 'Daniel Bugl' }
]
```

4. Next, we define the `App` component, and return a `<div>` container element, where we set some padding:

```
export default function App () {
    return (
        <div style={{ padding: 8 }}>
```

5. Now, we insert the `UserBar` and `CreatePost` components, passing the `user` prop to the `CreatePost` component:

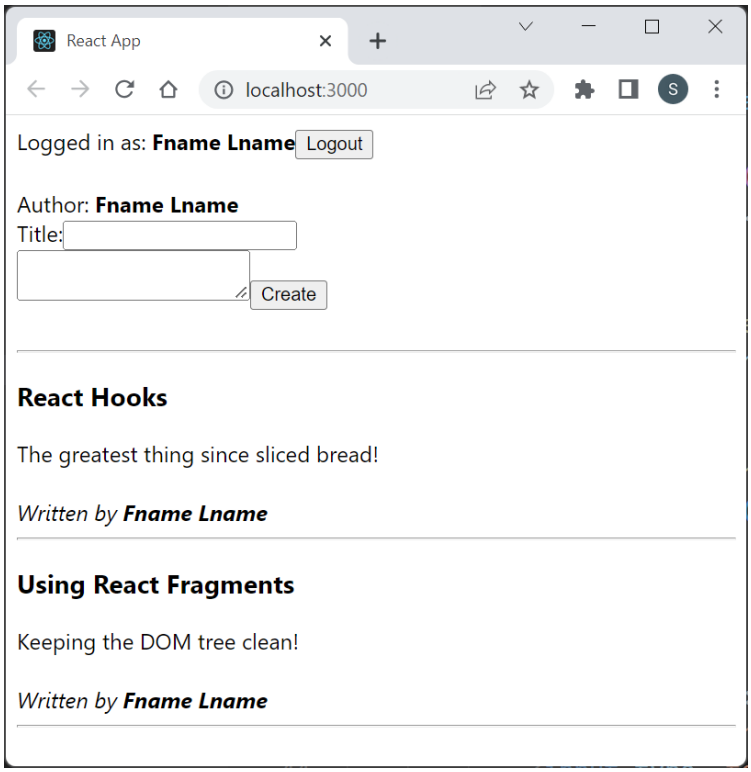
```
            <UserBar />
            <br />
            <CreatePost user={user} />
            <br />
            <hr />
```

Please note that you should always prefer spacing via CSS, rather than using the `<br />` HTML tag. However, at the moment, we are focusing on the UI, rather than its style, so we simply use HTML whenever possible.

6. Finally, we display the `PostList` component, listing all `posts`:

```
            <PostList posts={posts} />
        </div>
    )
}
```

7. After saving the file, `http://localhost:3000` should automatically refresh, and we can now see the full UI:



As we can see, all of the static components that we defined earlier are rendered together in one `App` component. Our app now looks just like the mock-up. Next, we can move on to making all of the components dynamic.

Exercise 4: Implementing stateful components with Hooks(chapter3_3)

Now that we have implemented the static structure of our application, we are going to add `useState` Hooks to it, in order to be able to handle state and dynamic interactions!

Adding Hooks for the users feature

To add Hooks for the users feature, we are going to have to replace the static `user` value with a State Hook. Then, we need to adjust the value when we log in, register and log out.

Step 1: Adjusting UserBar

Recall that when we created the `UserBar` component, we statically defined the `user` value. We are now going to replace this value with a State Hook!

Let's start modifying the `UserBar` component to make it dynamic:

1. Edit `src/user/UserBar.js`, and import the `useState` Hook by adjusting the `React` import statement, as follows:

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

2. Remove the following line of code:

   ```
   const user = 'Daniel Bugl'
   ```

   Replace it with a State Hook, using an empty user `''` as the default value:

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

3. Then, we pass the `setUser` function to the `Login`, `Register`, and `Logout` components:

   ```
   if (user) {
       return <Logout user={user} setUser={setUser} />
   } else {
        return (
           <React.Fragment>
               <Login setUser={setUser} />
               <Register setUser={setUser} />
           </React.Fragment>
       )
   }
   ```

Now, the `UserBar` component provides a `setUser` function, which can be used in the `Login`, `Register`, and `Logout` components to set or unset the `user` value.

Step 2: Adjusting the Login and Register components

In the `Login` and `Register` components, we need to use the `setUser` function to set the value of `user` accordingly, when we log in or register.

Login

In the `Login` component, we just ignore the **Password** field for now, and only process the **Username** field.

Let's start by modifying the `Login` component in order to make it dynamic:

1. Edit `src/user/Login.js`, and import the `useState` Hook:

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

2. Then, adjust the function definition to accept the `setUser` prop:

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

3. Now, we define a new State Hook for the value of the **Username** field:

```
    const [ username, setUsername ] = useState('')
```

4. Next, we define a handler function:

```
    function handleUsername (evt) {
        setUsername(evt.target.value)
     }
```

5. Then, we adjust the `input` field, in order to use the `username` value, and call the `handleUsername` function when the input changes:

```
            <input type="text" value={username}
onChange={handleUsername} name="login-username" id="login-username" />
```

6. Finally, we need to call the `setUser` function when the **Login** button is pressed, and thus the `form` is submitted:

```
            <form onSubmit={e => { e.preventDefault();
setUser(username) }} />
```

7. Additionally, we can disable the **Login** button when the `username` value is empty:

```
            <input type="submit" value="Login" disabled={username.length
=== 0} />
```

And it works—we can now enter a username, press the **Login** button, and then our `UserBar` component will change its state, and show the `Logout` component!

Register

For registration, we are additionally going to check whether the entered passwords are the same, and only then will we set the `user` value.

Let's start by modifying the `Register` component in order to make it dynamic:

1. First, we do the same steps as we did for `Login`, in order to handle the `username` field:

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

export default function Register ({ setUser }) {

    const [ username, setUsername ] = useState('')

    function handleUsername (evt)
{          setUsername(evt.target.value)
    }
    return (
        <form onSubmit={e => { e.preventDefault();
         setUser(username) }}>
            <label htmlFor="register-username">Username:</label>
            <input type="text" value={username}
onChange={handleUsername} name="register-username"
id="register-username" />
            <label htmlFor="register-password">Password:</label>
            <input type="password" name="register-password"
id="register-password" />
            <label htmlFor="register-password-repeat">Repeat
password:</label>
            <input type="password" name="register-password-repeat"
id="register-password-repeat" />
            <input type="submit" value="Register"
disabled={username.length === 0} />
        </form>
    )
}
```

2. Now, we define two new State Hooks for the **Password** and **Repeat password** fields:

```
    const [ password, setPassword ] = useState('')
    const [ passwordRepeat, setPasswordRepeat ] = useState('')
```

3. Then, we define two handler functions for them:

```
    function handlePassword (evt) {
        setPassword(evt.target.value)     }

    function handlePasswordRepeat (evt) {
        setPasswordRepeat(evt.target.value)     }
```

You might have noticed that we are always writing similar handler functions for `input` fields. Actually, this is the perfect use case for creating a custom Hook! We are going to learn how to do that in a future lab.

4. Next, we assign the `value` and `onChange` handler functions to the `input` fields:

```
                <label htmlFor="register-password">Password:</label>
                <input type="password" value={password}
onChange={handlePassword} name="register-password" id="register-
password" />
                <label htmlFor="register-password-repeat">Repeat
password:</label>
                <input type="password" value={passwordRepeat}
onChange={handlePasswordRepeat} name="register-password-repeat" id="register-
password-repeat" />
```

5. Finally, we check if the passwords match, and if they do not, we keep the button `disabled`:

```
                <input type="submit" value="Register"
disabled={username.length === 0 || password.length === 0 || password !==
passwordRepeat} />
```

And now we have successfully implemented a check on whether the passwords are equal, and we implemented registration!

Step 3: Adjusting Logout

There is still one thing missing for the users feature—we cannot log out yet.

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

1. Edit `src/user/Logout.js`, and add the `setUser` prop:

```
export default function Logout ({ user, setUser }) {
```

2. Then, adjust the `onSubmit` handler of `form` and set the user to `''`:

```
               <form onSubmit={e => { e.preventDefault(); setUser('')
}} />
```

As we are not creating a new Hook here, we do not need to import the `useState` Hook from React. We can simply use the `setUser` function passed to the `Logout` component as a prop.

Now, the `Logout` component sets the `user` value to `''` when we click on the **Logout** button.

Step 4: Passing the user to CreatePost

As you might have noticed, the `CreatePost` component still uses the hardcoded username. To be able to access the `user` value there, we need to move the Hook from the `UserBar` component, to the `App` component.
Let's refactor the definition of the `user` State Hook now:

1. Edit `src/user/UserBar.js`, and cut/remove the Hook definition that is there:

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

2. Then, we edit the function definition, and accept these two values as props:

```
export default function UserBar ({ user, setUser }) {
```

3. Now, we edit `src/App.js`, and import the `useState` Hook there:

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

4. Next, we remove the static `user` value definition:

```
const user = 'Daniel Bugl'
```

5. Then, we insert the `user` State Hook that we cut earlier into the `App` component function:

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

6. Now, we can pass `user` and `setUser` as props to the `UserBar` component:

```
<UserBar user={user} setUser={setUser} />
```

   The `user` state is a global state, so we are going to need it in many components across the app. At the moment, this means that we need to pass down the `user` value and the `setUser` function to each component that needs it. In a future lab, we are going to learn about React Context Hooks, which solve the problem of having to pass down props in such a way.

7. Finally, we only show the `CreatePost` component when the user is logged in. To do this, we use a pattern, which allows us to show a component based on a condition:

```
{user && <CreatePost user={user} />}
```

Now, the users feature is fully implemented—we can use the `Login` and `Register` components, and the `user` value also gets passed to the `CreatePost` component!

Exercise 5: Adding Hooks for the posts feature

After implementing the users feature, we are now going to implement the dynamic creation of posts. We do so by first adjusting the `App` component and then modifying the `CreatePost` component, in order to be able to insert new posts.
Let's get started by adjusting the App component.

Step 1: Adjusting the App component

As we know from the users feature, posts are also going to be global state, so we should define it in the `App` component.

Let's implement the `posts` value as global state now:

1. Edit `src/App.js`, and rename the current `posts` array to `defaultPosts`:

```
const defaultPosts = [
    { title: 'React Hooks', content: 'The greatest thing since
sliced bread!', author: 'Daniel Bugl' },
    { title: 'Using React Fragments', content: 'Keeping the DOM
tree clean!', author: 'Daniel Bugl' }
]
```

2. Then, define a new State Hook for the `posts` state:

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

3. Now, we pass the `posts` value and `setPosts` function as props to the `CreatePost` component:

```
            {user && <CreatePost user={user} posts={posts}
setPosts={setPosts} />}
```

Now, our `App` component provides the `posts` array, and a `setPosts` function to the `CreatePost` component. Let's move on to adjusting the CreatePost component.

Step 2: Adjusting the CreatePost component

Next, we need to use the `setPosts` function in order to insert a new post, when we press the **Create** button.
Let's start modifying the `CreatePost` component in order to make it dynamic:

1. Edit `src/posts/CreatePost.js`, and import the `useState` Hook:

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

2. Then, adjust the function definition to accept the `posts` and `setPosts` props:

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

3. Next, we define two new State Hooks—one for the `title` value, and one for the `content` value:

```
const [ title, setTitle ] = useState('')
const [ content, setContent ] = useState('')
```

4. Now, we define two handler functions—one for the `input` field, and one for the `textarea`:

```
function handleTitle (evt) {
    setTitle(evt.target.value)
}

function handleContent (evt) {
    setContent(evt.target.value)
}
```

5. We also define a handler function for the **Create** button:

```
function handleCreate () {
```

6. In this function, we first create a `newPost` object from the `input` field values:

```
    const newPost = { title, content, author: user }
```

In newer JavaScript versions, we can shorten the following object assignment: `{ title: title }`, to `{ title }`, and it will have the same effect. So, instead of doing `{ title: title, contents: contents }`, we can simply do `{ title, contents }`.

7. Then, we set the new `posts` array by first adding `newPost` to the array, then using the spread syntax to list all of the existing `posts`:

```
    setPosts([ newPost, ...posts ])
}
```

8. Next, we add the `value` and handler functions to the `input` field and `textarea` element:

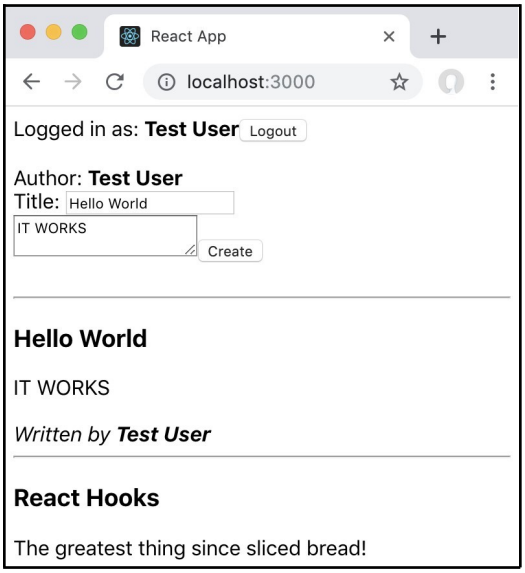
```
        <div>
            <label htmlFor="create-title">Title:</label>
            <input type="text" value={title}
onChange={handleTitle} name="create-title"
                id="create-title" />
        </div>
        <textarea value={content} onChange={handleContent} />
```

Usually in HTML, we put the value of `textarea` as its children. However, in React, `textarea` can be handled like any other `input` field, by using the `value` and `onChange` props.

9. Finally, we pass the `handleCreate` function to the `onSubmit` handler of the `form` element:

```
    <form onSubmit={e => { e.preventDefault(); handleCreate()
}}>
```

10. Now, we can log in and create a new post, and it will be inserted at the beginning of the feed:



Our first version of the blog app using Hooks, after inserting a new blog post

As we can see, now our application is fully dynamic, and we can use all of its features!