5 Implementing React Context

Exercise 1: Introducing React context

In the previous labs, we passed down the `user` state and `dispatch` function from the `App` component, to the `UserBar` component; and then from the `UserBar` component to the `Logout`, `Login`, and `Register` components. React context provides a solution to this cumbersome way of passing down props over multiple levels of components, by allowing us to share values between components, without having to explicitly pass them down via props. As we are going to see, React context is perfect for sharing values across the whole application.

First, we are going to have a closer look at the problem of passing down props. Then, we are going to introduce React context as a solution to the problem.

Step 1: Passing down props

Before learning about React context in depth, let's recap what we implemented in the earlier chapters, in order to get a feeling for the problem that contexts solve:

1. In `src/App.js`, we defined the `user` state and the `dispatch` function:

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

2. Then, we passed the `user` state and the `dispatch` function to the `UserBar` component (and the `CreatePost` component):

```
    return (
        <div style={{ padding: 8 }}>
            <UserBar user={user} dispatch={dispatch} />
            <br />
            {user && <CreatePost user={user} posts={posts}
dispatch={dispatch} />}
            <br />
            <hr />
            <PostList posts={posts} />
        </div>
    )
```

3. In the `src/user/UserBar.js` component, we took the `user` state as a prop, and then passed it down to the `Logout` component. We also took the `dispatch` function as a prop, and passed it to the `Logout`, `Login`, and `Register` components:

```
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>
        )
    }
}
```

4. Finally, we used the `dispatch` and `user` props in the `Logout`, `Login`, and `Register` components.

React context allows us to skip steps 2 and 3, and jump straight from step 1 to step 4. As you can imagine, with larger apps, context becomes even more useful, because we might have to pass down props over many levels.

Exercise 2: Introducing React context

React context is used to share values across a tree of React components. Usually, we want to share global values, such as the `user` state and the `dispatch` function, the theme of our app, or the chosen language.

React context consists of two parts:

- The **provider**, which provides (sets) the value
- The **consumer**, which consumes (uses) the value

We are first going to look at how contexts work, using a simple example, and, in the next section, we are going to implement them in our blog app. We create a new project with the `create-react-app` tool. In our simple example, we are going to define a theme context, containing the primary color of an app.

Step 1: Defining the context

First, we have to define the context. The way this works has not changed since Hooks were introduced.

We simply use the `React.createContext(defaultValue)` function to create a new context object. We set the default value to `{ primaryColor: 'deepskyblue' }`, so our default primary color, when no provider is defined, will be `'deepskyblue'`.

In `src/App.js`, add the following definition before the `App` function:

```
export const ThemeContext = React.createContext({ primaryColor: 'deepskyblue' })
```

Note how we are exporting `ThemeContext` here, because we are going to need to import it for the consumer.

That is all we need to do to define a context with React. Now we just need to define the consumer.

Step 2: Defining the consumer

Now, we have to define the consumer in our `Header` component. We are going to do this in the traditional way for now, and in the next steps use Hooks to define the consumer:

1. Create a new **`src/Header.js`** file
2. First, we have to import `ThemeContext` from the `App.js` file:

```
import React from 'react'
import { ThemeContext } from './App'
```

3. Now, we can define our component, where we use the `ThemeContext.Consumer` component and a `render` function as `children` prop, in order to make use of the context value:

```
const Header = ({ text }) => (
    <ThemeContext.Consumer>
        {theme => (
```

4. Inside the `render` function, we can now make use of the context value to set the `color` style of our `Header` component:

```
                <h1 style={{ color: theme.primaryColor }}>{text}</h1>
            )}
    </ThemeContext.Consumer>
)
 export default Header
```

5. Now, we still need to import the `Header` component in `src/App.js`, by adding the following `import` statement:

```
import Header from './Header'
```

6. Then, we replace the current `App` function with the following code:

```
const App = () => (
    <Header text="Hello World" />
)

export default App
```

Using contexts like this works, but, as we have learned in the first chapter, using components with `render` function props in this way clutters our UI tree, and makes our app harder to debug and maintain.

Step 3: Using Hooks

A better way to use contexts is with the `useContext` Hook! That way, we can use context values like any other value, in a similar way to the `useState` Hook:

1. Edit `src/Header.js`. First, we import the `useContext` Hook from React, and the `ThemeContext` object from `src/App.js`:

```
import React, { useContext } from 'react' import
{ ThemeContext } from './App'
```

2. Then, we create our `Header` component, where we now define the `useContext` Hook:

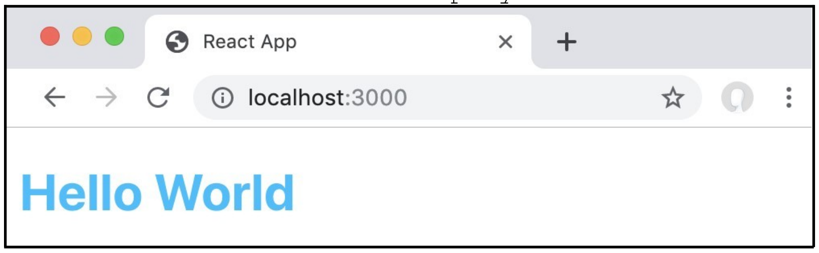
```
const Header = ({ text }) => {
      const theme = useContext(ThemeContext)
```

3. The rest of our component will be the same as before, except that, now, we can simply return our `Header` component, without using an additional component for the consumer:

```
      return <h1 style={{ color: theme.primaryColor }}>{text}</h1> } export default
Header
```

As we can see, using Hooks makes our context consumer code much more concise. Furthermore, it will be easier to read, maintain, and debug.

We can see that the header now has the color `deepskyblue`:



A simple app with a Context Hook!

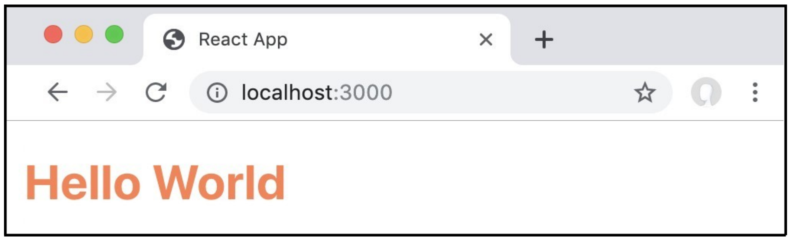
As we can see, our theme context successfully provides the theme for the header.

Step 4: Defining the provider

Contexts use the default value that is passed to `React.createContext`, when there is no provider defined. This is useful for debugging the components when they are not embedded in the app. For example, we could debug a single component as a standalone component. In an app, we usually want to use a provider to provide the value for the context, which we are going to define now. Edit `src/App.js`, and in our `App` function, we simply wrap the `Header` component with a `<ThemeContext.Provider>` component, where we pass `coral` as `primaryColor`:

```
const App = () => (
    <ThemeContext.Provider value={{ primaryColor: 'coral' }}>
        <Header text="Hello World" />
    </ThemeContext.Provider>
)

export default App
```

We can now see that our header color changed from `deepskyblue` to `coral`:



Hello World

Our provider changed the color of the header

If we want to change the value of our context, we can simply adjust the `value` prop that is passed to the `Provider` component.

> Please note that the default value of a context is not used when we define a provider without passing the `value` prop to it! If we define a provider without a `value` prop, then the value of the context will be `undefined`.

Now that we have defined a single provider for our context, let's move on to defining multiple, nested providers.

Step 5: Nested providers

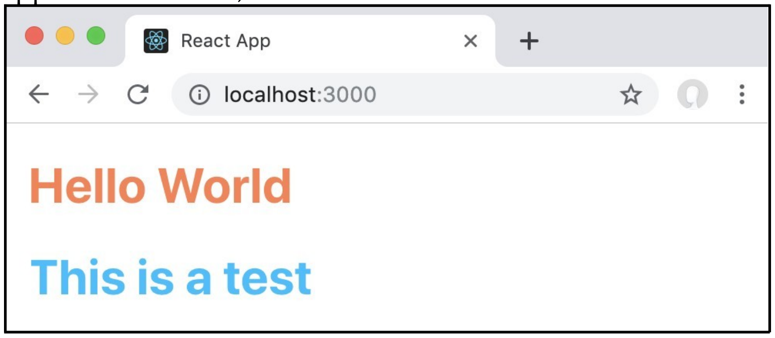
With React context, it is also possible to define multiple providers for the same context. Using this technique, we can override the context value in certain parts of our app. Let's consider the earlier example, and add a second header to it: 1. Edit `src/App.js`, and add a second `Header` component:

```
const App = () => (
    <ThemeContext.Provider value={{ primaryColor: 'coral' }}>
        <Header text="Hello World" />
        <Header text="This is a test" />
    </ThemeContext.Provider>
)

export default App
```

2. Now, define a second `Provider` component with a different `primaryColor`:

```
const App = () => (
    <ThemeContext.Provider value={{ primaryColor: 'coral' }}>
        <Header text="Hello World" />
        <ThemeContext.Provider value={{ primaryColor: 'deepskyblue'
}}>
            <Header text="This is a test" />
        </ThemeContext.Provider>
    </ThemeContext.Provider>
)

export default App
```

If we open the app in our browser, the second header now has a different color from the first one:



Hello World

This is a test

Overriding context values with nested providers

As we can see, we can override React context values by defining providers. Providers can also be nested, therefore overriding the values of other providers that are higher up in the component tree.

Exercise 3: Implementing themes(chapter5_2)

After learning how to implement themes in a small example, we are now going to implement themes in our blog app, using React context and Hooks.

Step 1: Defining the context

First, we have to define the context. Instead of defining it in the `src/App.js` file, in our blog app, we are going to create a separate file for the context. Having a separate file for contexts makes it easier to maintain them later on. Furthermore, we always know where to import the contexts from, because it is clear from the filename.

Let's start defining a theme context:

1. Create a new `src/contexts.js` file.
2. Then, we import `React`:

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

3. Next, we define the `ThemeContext`. As before in our small example, we set the default `primaryColor` to `deepskyblue`. Additionally, we set the `secondaryColor` to `coral`:

   ```
   export const ThemeContext = React.createContext({
       primaryColor: 'deepskyblue',
       secondaryColor: 'coral'
   })
   ```

Now that we have defined our context, we can move on to defining the Context Hooks.

Defining the Context Hooks

After defining the context, we are going to define our consumers, using Context Hooks. We start by creating a new component for the header, then define a Context Hook for our existing `Post` component.

Step 2: Creating the Header component

First, we create a new `Header` component, which is going to display `React Hooks Blog` in the `primaryColor` of our app.
Let's create the `Header` component now:

1. Create a new **`src/Header.js`** file.
2. In this file, we import `React`, and the `useContext` Hook:

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

3. Next, we import the `ThemeContext` from the previously created `src/contexts.js` file:

   ```
       import { ThemeContext } from `'./contexts'
   ```

4. Then, we define our `Header` component, and the Context Hook. Instead of storing the context value in a `theme` variable, we use destructuring to directly extract the `primaryColor` value:

   ```
   const Header = ({ text }) => {
       const { primaryColor } = useContext(ThemeContext)
   ```

5. Finally, we return the `h1` element, as we did before in our small example, and `export` the `Header` component:

   ```
       return <h1 style={{ color: primaryColor }}>{text}</h1>
   }
   export default Header
   ```

Now our `Header` component is defined, and we can use it.

Step 3: Using the Header component

After creating the `Header` component, we are going to use it in the `App` component, as follows:

1. Edit `src/App.js`, and import the `Header` component:

   ```
   import Header from './Header'
   ```

2. Then, render the `Header` component before the `UserBar` component:

   ```
       return (
           <div style={{ padding: 8 }}>
               <Header text="React Hooks Blog" />
               <UserBar user={user} dispatch={dispatch} />
   ```

You might want to refactor the `React Hooks Blog` value into a prop that is passed to the `App` component (app config), because we are already using it three times in this component.

Now, our `Header` component will be rendered in the app and we can move on to implementing the Context Hook in the Post component.

Step 4: Implementing the Context Hook for the Post component

Next, we want to display the `Post` headers in the secondary color. To do this, we need to define a Context Hook for the `Post` component, as follows:

1. Edit `src/post/Post.js`, and adjust the `import` statement to import the `useContext` Hook:

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

2. Next, we import the `ThemeContext`:

    ```
    import { ThemeContext } from '../contexts'
    ```

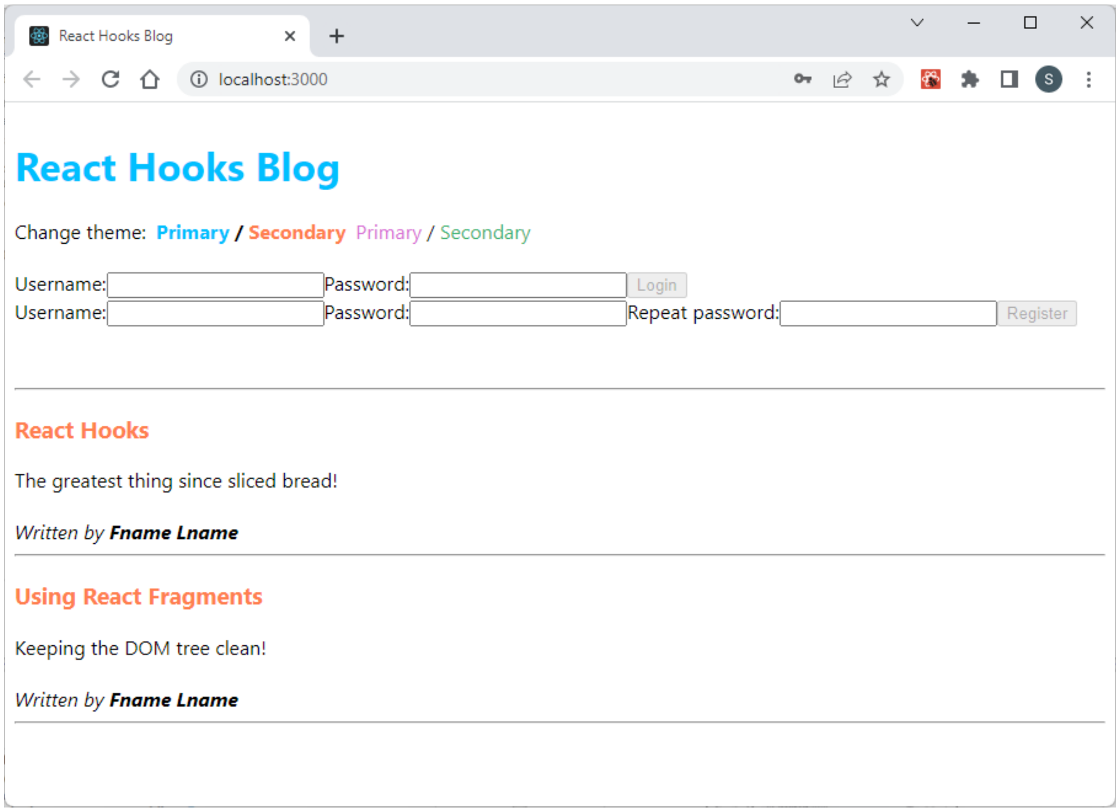
3. Then, we define a Context Hook in the `Post` component, and get the `secondaryColor` value from the theme, via destructuring:

    ```
    export default function Post ({ title, content, author }) {
        const { secondaryColor } = useContext(ThemeContext)
    ```

4. Finally, we use the `secondaryColor` value to style our `h3` element:

    ```
    return (
        <div>
            <h3 style={{ color: secondaryColor }}>{title}</h3>
    ```

If we look at our app now, we can see that both colors are used properly from the `ThemeContext`:



React Hooks Blog

Change theme: Primary / Secondary Primary / Secondary

Username: Password: Login
Username: Password: Repeat password: Register

React Hooks

The greatest thing since sliced bread!

Written by **Fname Lname**

Using React Fragments

Keeping the DOM tree clean!

Written by **Fname Lname**

As we can see, our app now uses the primary color for the main header, and the secondary color for the post titles.

Step 5: Defining the provider

Right now, our Context Hooks use the default value that is specified by the context, when no provider is defined. To be able to change the value, we need to define a provider.

Let's start defining the provider:

1. Edit **`src/App.js`**, and import the `ThemeContext`: `import`

   `{ ThemeContext } from './contexts'`

2. Wrap the whole app with the `ThemeContext.Provider` component, providing the same theme that we set as the default value earlier:

```
return (
    <ThemeContext.Provider value={{ primaryColor:
      'deepskyblue', secondaryColor: 'coral' }}>
        <div style={{ padding: 8 }}>
            <Header text="React Hooks Blog" />
            ...
            <PostList posts={posts} />
        </div>
    </ThemeContext.Provider>
)
```

Our app should look exactly the same way as before, but now we are using the value from the provider!

Exercise 4: Dynamically changing the theme

Now that we have defined a provider, we can use it to dynamically change the theme. Instead of passing a static value to the provider, we are going to use a State Hook that defines the current theme. Then, we are going to implement a component that changes the theme.

Step 1: Using a State Hook with the context provider

First, we are going to define a new State Hook, which we are going to use to set the value for the context provider.

Let's define a State Hook, and use it in the context provider: 1. Edit **`src/App.js`**, and import the `useState` Hook:

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

2. Define a new State Hook at the beginning of the `App` component; here we set the default value to our default theme:

```
export default function App () {
    const [ theme, setTheme ] = useState({
        primaryColor: 'deepskyblue',
        secondaryColor: 'coral'
    })
```

3. Then, we pass the `theme` value to the `ThemeContext.Provider` component:

```
return (
    <ThemeContext.Provider value={theme}>
```

Our app is still going to look the same way as before, but we are now ready to dynamically change our theme!

Step 2: Implementing the ChangeTheme component

The final part of our theme feature is a component that can be used to change the theme dynamically, by making use of the State Hook that we defined earlier. The State Hook is going to re-render the `App` component, which will change the value that is passed to the `ThemeContext.Provider`, which, in turn, is going to re-render all the components that make use of the `ThemeContext` Context Hook.

Let's start implementing the `ChangeTheme` component:

1. Create a new **`src/ChangeTheme.js`** file.
2. As always, we have to import `React` first, before we can define a component:

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

3. In order to be able to easily add new themes later on, we are going to create a constant `THEMES` array, instead of manually copying and pasting the code for the different themes. This is going to make our code much more concise, and easier to read:

```
const THEMES = [
    { primaryColor: 'deepskyblue', secondaryColor: 'coral' },
    { primaryColor: 'orchid', secondaryColor: 'mediumseagreen' }
]
```

It is a good idea to give constant values that are hardcoded a special name, such as writing the whole variable name in caps. Later on, it might make sense to put all these configurable hardcoded values in a separate `src/config.js` file.

4. Next, we define a component to render a single `theme`:

```
    function ThemeItem ({ theme, active, onClick }) {
```

5. Here, we render a link, and display a small preview of the theme, by showing the **Primary** and **Secondary** colors:

```
    return (
        <span onClick={onClick} style={{ cursor: 'pointer', paddingLeft:
8, fontWeight: active ? 'bold' : 'normal' }}>
            <span style={{ color: theme.primaryColor
}}>Primary</span> / <span style={{ color: theme.secondaryColor
}}>Secondary</span>
        </span>
    )
}
```

Here, we set the cursor to `pointer`, in order to make the element appear clickable. We could also use an `<a>` element; however, this is not recommended if we do not have a valid link target, such as a separate page.

6. Then, we define the `ChangeTheme` component, which accepts the `theme` and `setTheme` props:

```
    export default function ChangeTheme ({ theme, setTheme }) {
```

7. Next, we define a function to check if a theme object is the currently active theme:

```
        function isActive (t) {
            return t.primaryColor === theme.primaryColor &&
              t.secondaryColor === theme.secondaryColor
        }
```

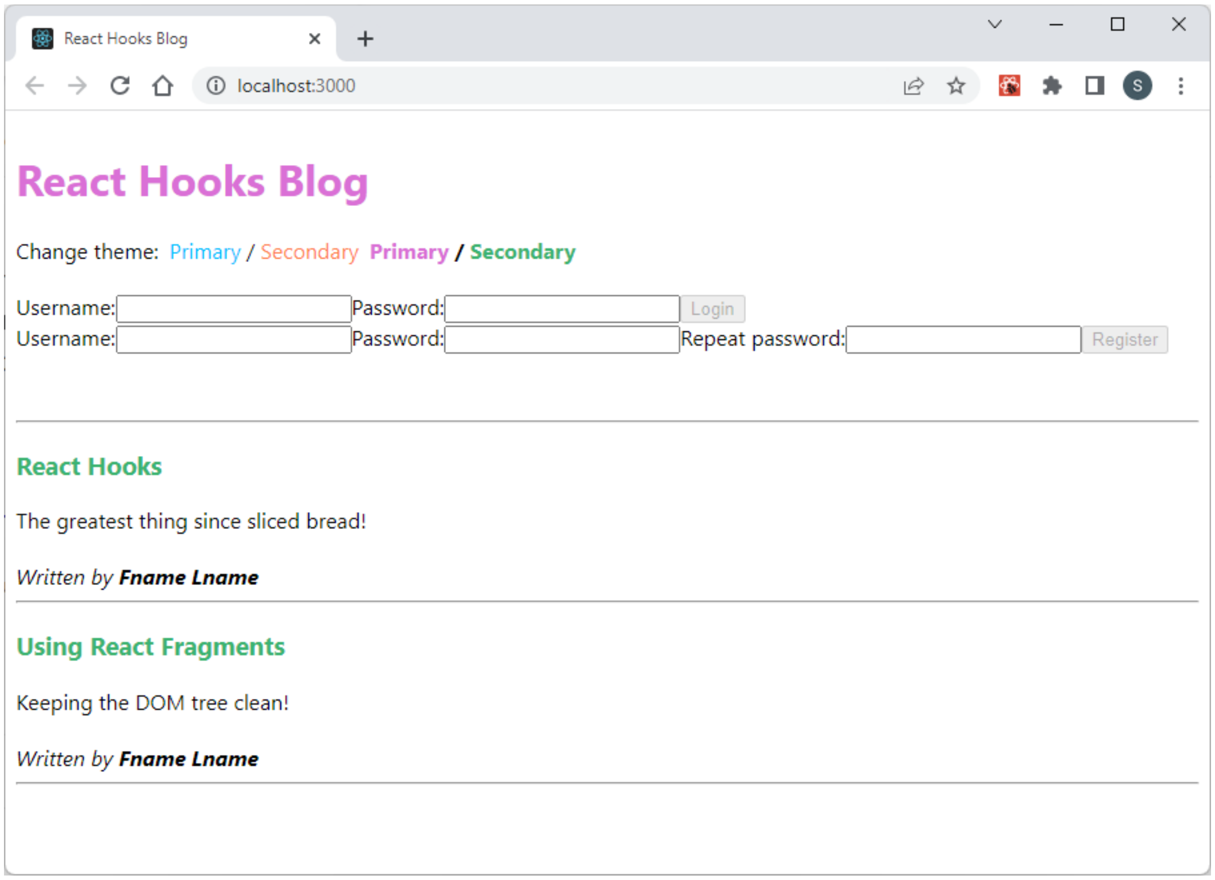
8. Now, we use the `.map` function to render all of the available themes, and call the `setTheme` function when clicking on them:

```
        return (
            <div>
                Change theme:
                {THEMES.map((t, i) =>
                    <ThemeItem key={'theme-' + i} theme={t}
       active={isActive(t)} onClick={() => setTheme(t)} />
                )}
            </div>
        )
    }
```

9. Finally, we can import and render the `ChangeTheme` component, after the `Header` component in `src/App.js`:

```
    import ChangeTheme from './ChangeTheme'
    // ...
        return (
            <ThemeContext.Provider value={theme}>
                <div style={{ padding: 8 }}>
                    <Header text="React Hooks Blog" />
                    <ChangeTheme theme={theme} setTheme={setTheme} />
                    <br />
```

As we can see, we now have a way to change the theme in our app:


React Hooks Blog

localhost:3000

React Hooks Blog

Change theme: Primary / Secondary **Primary / Secondary**

Username: Password: Login
Username: Password: Repeat password: Register

React Hooks

The greatest thing since sliced bread!

*Written by **Fname Lname***

Using React Fragments

Keeping the DOM tree clean!

*Written by **Fname Lname***

Our app after changing the theme, using Context Hooks in combination with a State Hook

Now, we have a context that is consumed via Hooks, which can also be changed via Hooks!

Exercise 5: Using context for global state (chapter5_3)

After learning how to use React context to implement themes in our blog app, we are now going to use a context to avoid having to manually pass down the `state` and `dispatch` props for our global app state.

Step 1: Defining StateContext

We start by defining the context in our `src/contexts.js` file.

In **`src/contexts.js`**, we define the `StateContext`, which is going to store the `state` value and the `dispatch` function:

```
export const StateContext = React.createContext({
    state: {},
    dispatch: () => {}
})
```

We initialized the `state` value as an empty object, and the `dispatch` function as an empty function, which will be used when no provider is defined.

Step 2: Defining the context provider

Now, we are going to define the context provider in our `src/App.js` file, which is going to get the values from the existing Reducer Hook.
Let's define the context provider for global state now:

1. In `src/App.js`, import the `StateContext` by adjusting the existing `import` statement:

   ```
   import { ThemeContext, StateContext } from './contexts'
   ```

2. Then, we define a new context provider, by returning it from our `App` function:

   ```
   return (
       <StateContext.Provider value={{ state, dispatch }}>
           <ThemeContext.Provider value={theme}>
               ...
           </ThemeContext.Provider>
       </StateContext.Provider>
   )
   ```

Now, our context provider provides the `state` object and the `dispatch` function to the rest of our app, and we can move on to consuming the context value.

Step 3: Using StateContext

Now that we have defined our context and provider, we can use the `state` object and the `dispatch` function in various components.

We start by removing the props that we manually passed to our components in `src/App.js`. Delete the following code segments marked in bold:

```
<div style={{ padding: 8 }}>
    <Header text="React Hooks Blog" />
    <ChangeTheme theme={theme} setTheme={setTheme} />
    <br />
    <UserBar user={user} dispatch={dispatch} />
    <br />
    {user && <CreatePost user={user} posts={posts}
         dispatch={dispatch} />}
    <br />
    <hr />
    <PostList posts={posts} />
</div>
```

As we are using contexts, there is no need to pass down props manually anymore. We can now move on to refactoring the components.

Step 4: Refactoring user components

First, we refactor the user components, and then we move on to the post components.

Let's refactor the user-related components now:

1. Edit `src/user/UserBar.js`, and also remove the props there (code marked in bold should be removed), since we do not need to manually pass them down anymore:

```
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>
        )
    }
}
```

2. Then, we import the `useContext` Hook and the `StateContext` in `src/user/UserBar.js`, in order to be able to tell whether the user is logged in or not:

```
import React, { useContext } from 'react'
import { StateContext } from '../contexts'
```

3. Now, we can use the Context Hook to get the `user` state from our `state` object:

```
export default function UserBar () {
    const { state } = useContext(StateContext)
    const { user } = state
```

4. Again, we import `useContext` and `StateContext` in `src/user/Login.js`:

```
import React, { useState, useContext } from 'react'
import { StateContext } from '../contexts'
```

5. Then, we remove the `dispatch` prop, and use the Context Hook instead:

```
export default function Login () {
    const { dispatch } = useContext(StateContext)
```

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

```
import React, { useState, useContext } from 'react' import
{ StateContext } from '../contexts'

export default function Register () {
     const { dispatch } = useContext(StateContext)
```

7. In the `src/user/Logout.js` component, we do the same, but also get the `user` state from the `state` object:

```
import React, { useContext } from 'react' import
{ StateContext } from '../contexts'

export default function Logout () {
    const { state, dispatch } = useContext(StateContext)
    const { user } = state
```

Our user-related components now use a context instead of props. Let's move on to refactoring the post-related components.

Step 5: Refactoring post components

Now, all that is left to do is refactoring the post components; then our whole app will be using React context for global state:

1. We start with the `src/post/PostList.js` component, where we import `useContext` and `StateContext`, remove the props, and use the Context Hook instead:

   ```
   import React, { useContext } from 'react'
   import { StateContext } from '../contexts'

   import Post from './Post'

   export default function PostList () {
       const { state } = useContext(StateContext)
       const { posts } = state
   ```

2. We do the same for the `CreatePost` component, which is the last component that we need to refactor:

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

   import { StateContext } from '../contexts'

   export default function CreatePost () {
       const { state, dispatch } = useContext(StateContext)
       const { user } = state
   ```

Our app works in the same way as before, but now we use a context for global state, which makes our code much cleaner, and avoids having to pass down props!