7 Using Hooks for Routing

Exercise 1: Creating multiple pages

At the moment, our blog application is a so-called single-page application. However, most larger apps consist of multiple pages. In a blog app, we at least want to have a separate page for each blog post.

Before we can set up routing, we need to create the various pages that we want to render. In our blog app, we are going to define the following pages:

- A home page, which will display a list of all posts
- A post page, which will display a single post

All pages will show a `HeaderBar`, which renders the `Header`, `UserBar`, `ChangeTheme`, and `CreatePost` components. We are now going to start by creating a component for the `HeaderBar`. Afterward, we are going to implement the page components.

Creating the HeaderBar component

First of all, we are going to refactor some contents of our `App` component into a `HeaderBar` component. The `HeaderBar` component will contain everything that we want to display on every page: the `Header`, `UserBar`, `ChangeTheme`, and `CreatePost` components.

Step 1: Let's start creating the `HeaderBar` component:

1. Create a new folder: `src/pages/`.
2. Create a new file, **`src/pages/HeaderBar.js`**, import `React` (with the `useContext` Hook), and define the component there. It will accept the `setTheme` function as prop:

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

export default function HeaderBar ({ setTheme }) {
   return (
        <div>
        </div>
    )
}
```

3. Now, cut the following code from the `src/App.js` component, and insert it between the `<div>` tags of the `HeaderBar` component:

```
<Header text="React Hooks Blog" />
<ChangeTheme theme={theme} setTheme={setTheme} />
<br />
<React.Suspense fallback={"Loading..."}>
    <UserBar />
</React.Suspense>
<br />
{user && <CreatePost />}
```

4. Also, cut the following import statements (and adjust the paths) from `src/App.js` and insert them at the beginning of the **`src/pages/HeaderBar.js`** file, after the `import React from 'react'` statement:

```
import CreatePost from '../post/CreatePost'
import UserBar from '../user/UserBar'
import Header from '../Header'
import ChangeTheme from '../ChangeTheme'
```

5. Additionally, import the `ThemeContext` and the `StateContext`:

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

6. Then, define two Context Hooks for the `theme` and `state`, and pull the `user` variable out of the `state` object in `src/pages/HeaderBar.js`, as we need it for a conditional check to determine whether we should render the `CreatePost` component:

```
export default function HeaderBar ({ setTheme }) {
    const theme = useContext(ThemeContext)
    const { state } = useContext(StateContext)
    const { user } = state

    return (
```

7. Now, we import the `HeaderBar` component in `src/App.js`:

```
import HeaderBar from './pages/HeaderBar'
```

8. Finally, we render the `HeaderBar` component in `src/App.js`:

```
        <div style={{ padding: 8 }}>
            <HeaderBar setTheme={setTheme} />
            <hr />
```

Now, we have a separate component for the `HeaderBar`, which will be shown on all pages. Next, we move on to creating the `HomePage` component.

Creating the HomePage component

Now, we are going to create the `HomePage` component from the `PostList` component and the Resource Hook that is concerned with the posts. Again, we are going to refactor `src/App.js`, in order to create a new component.

Step 2: Let's start creating the `HomePage` component:

1. Create a new file, `src/pages/HomePage.js`, import `React` with the `useEffect` and `useContext` Hooks, and define the component there. We also define a Context Hook and pull out the `state` object and `dispatch` function:

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

import { StateContext } from '../contexts'

export default function HomePage () {
    const { state, dispatch } = useContext(StateContext)

    const { error } = state

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

2. Then, cut the following import statements (and adjust the paths) from `src/App.js`, and add them after the `import React from 'react'` statement in `src/pages/HomePage.js`:

```
import { useResource } from 'react-request-hook'
import PostList from '../post/PostList'
```

3. Next, cut the following Hook definitions from `src/App.js`, and insert them before the `return` statement of the `HomePage` function:

```
    const [ posts, getPosts ] = useResource(() => ({
        url: '/posts',
        method: 'get'
    }))
    useEffect(getPosts, [])
    useEffect(() => {
        if (posts && posts.error) {
            dispatch({ type: 'POSTS_ERROR' })
        }
        if (posts && posts.data) {
            dispatch({ type: 'FETCH_POSTS', posts:
posts.data.reverse() })
        }
    }, [posts])
```

4. Now, cut the following rendered code from `src/App.js`, and insert it in between the `<div>` tags of `src/pages/HomePage.js`:

```
            {error && <b>{error}</b>}
            <PostList />
```

5. Then, import the `HomePage` component in `src/App.js`:

```
import HomePage from './pages/HomePage'
```

6. Finally, render the `HomePage` component below the `<hr />` tag:

```
            <hr />
            <HomePage />
```

Now, we have successfully refactored our current code into a `HomePage` component. Next, we move on to creating the `PostPage` component.

Creating the PostPage component

We are now going to define a new page component, where we will only fetch a single post from our API and display it.

Step 3: Let's start creating the `PostPage` component now:

1. Create a new **`src/pages/PostPage.js`** file.
2. Import `React`, the `useEffect` and `useResource` Hooks and the `Post` component:

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

import { useResource } from 'react-request-hook'

import Post from '../post/Post'
```

3. Now, define the `PostPage` component, which is going to accept the post `id` as prop:

```
export default function PostPage ({ id }) {
```

4. Here, we define a Resource Hook that will fetch the corresponding `post` object. We pass the `id` as dependency to the Effect Hook so that our resource re-fetches when the `id` changes:

```
const [ post, getPost ] = useResource(() => ({
    url: `/posts/${id}`,
    method: 'get'
}))
useEffect(getPost, [id])
```

5. Finally, we render the `Post` component:

```
return (
    <div>
        {(post && post.data)
            ? <Post {...post.data} />
            : 'Loading...'
        }
        <hr />
    </div>
)
}
```

We now also have a separate page for single posts.

Step 4: Testing out the PostPage

To test out the new page, we are going to replace the `HomePage` component in `src/App.js` with the `PostPage` component, as follows: 1. Import the `PostPage` component in **`src/App.js`**:

```
import PostPage from './pages/PostPage'
```

2. Now, replace the `HomePage` component with the `PostPage` component:

```
<PostPage id={'react-hooks'} />
```

As we can see, now only one post, the **React Hooks** post, gets rendered.

Exercise 2: Implementing routing(Chapter7_2)

We are going to use the Navi library for routing. Navi supports React Suspense, Hooks, and error boundary APIs of React natively, which makes it the perfect fit to implement routing through the use of Hooks. To implement routing, we are first going to define routes from the pages that we defined in the previous section. Finally, we are going to define links from the main page to the corresponding post pages, and from these pages back to the main page.

Toward the end of this chapter, we are going to extend our routing functionality by implementing routing Hooks.

Step 1: Defining routes

The first step when implementing routing is to install the `navi` and `react-navi` libraries. Then, we define the routes. Follow the given steps to do so:

1. First, we have to install the libraries using `npm`:

```
> npm install --save navi react-navi
```

2. Then, in `src/App.js`, we import the `Router` and `View` components and the `mount` and `route` functions from the Navi library:

```
import { Router, View } from 'react-navi'
import { mount, route } from 'navi'
```

3. Make sure that the `HomePage` component is imported:

```
import HomePage from './pages/HomePage'
```

4. Now, we can define the `routes` object using the `mount` function:

```
const routes = mount({
```

5. In this function, we define our routes, starting with the main route:

```
    '/': route({ view: <HomePage /> }),
```

6. Next, we define the route for a single post, here we use URL parameters (`:id`), and a function to dynamically create the `view`:

```
    '/view/:id': route(req => {
        return { view: <PostPage id={req.params.id} /> }
    }),
})
```

7. Finally, we wrap our rendered code with the `<Router>` component, and replace the `<PostPage>` component with the `<View>` component in order to dynamically render the current page:

```
<Router routes={routes}>
    <div style={{ padding: 8 }}>
        <HeaderBar setTheme={setTheme} />
        <hr />
        <View />
    </div>
</Router>
```

Now, if we go to `http://localhost:3000`, we can see a list of all posts, and when we go to `http://localhost:3000/view/react-hooks`, we can see a single post: the **React Hooks** post.

Defining links

Now, we are going to define links from each post to the page of the corresponding single post, and then back to the main page from the post page. The links will be used to access the various routes that have been defined in our app. First, we are going to define links from the home page to the single post pages. Next, we are going to define links from the single post pages back to the main page.

Step 2: Defining links to the posts

We start by shortening the post `content` in the list, and defining links from the `PostList` to the corresponding post pages. To do so, we have to define static links from the `PostList` on the home page to the specific post pages.

Let's define those links now:

1. Edit **`src/post/Post.js`**, and import the `Link` component from `react-navi`:

```
import { Link } from 'react-navi'
```

2. Then, we are going to add two new props to the `Post` component: `id` and `short`, which will be set to `true` when we want to display the shortened version of the post. Later, we are going to set `short` to `true` in the `PostList` component:

```
function Post ({ id, title, content, author, short = false }) {
```

3. Next, we are going to add some logic to trim the post `content` to `30` characters when listing it:

```
let processedContent = content
if (short) {
    if (content.length > 30) {
        processedContent = content.substring(0, 30) + '...'
    }
}
```

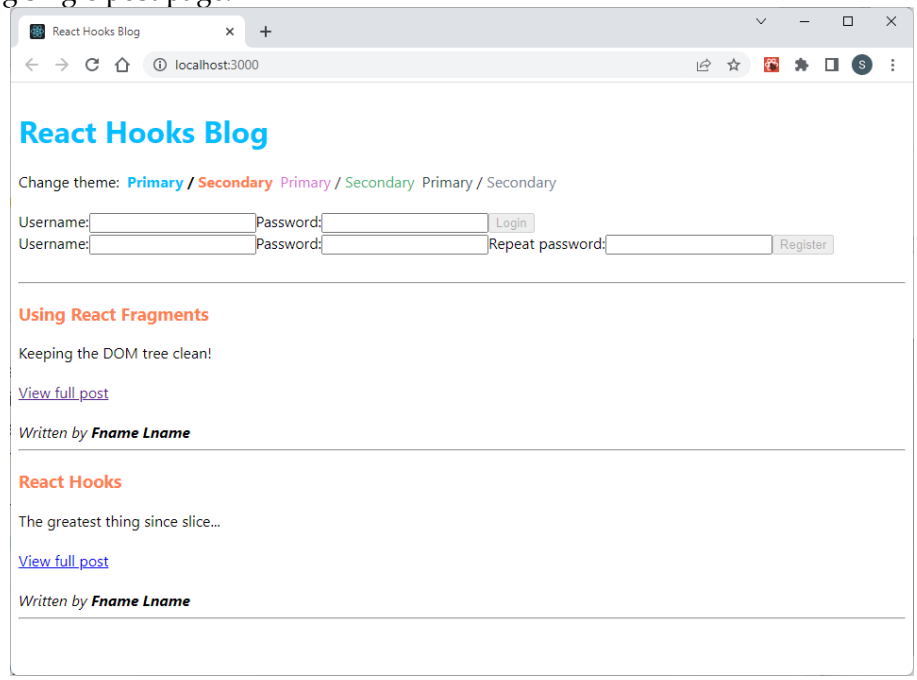
4. Now, we can display the `processedContent` value instead of the `content` value, and a `Link` to view the full post:

```
<div>{processedContent}</div>
{short &&
    <div>
        <br />
        <Link href={`/view/${id}`}>View full
          post</Link>
    </div>
}
```

5. Finally, we set the `short` prop to `true` within the `PostList` component. Edit `src/post/PostList.js`, and adjust the following code:

```
<Post {...p} short={true} />
```

Now we can see that each post on the main page is trimmed to `30` characters, and has a link to the corresponding single post page:



Displaying a link in the PostList

As we can see, routing is quite simple. Now, each post has a link to its corresponding full post page.

Step 3: Defining the links to the main page

Now, we just need a way to get back to the main page from a single post page. We are going to repeat a similar process to what we have done previously. Let's define the links back to the main page now:

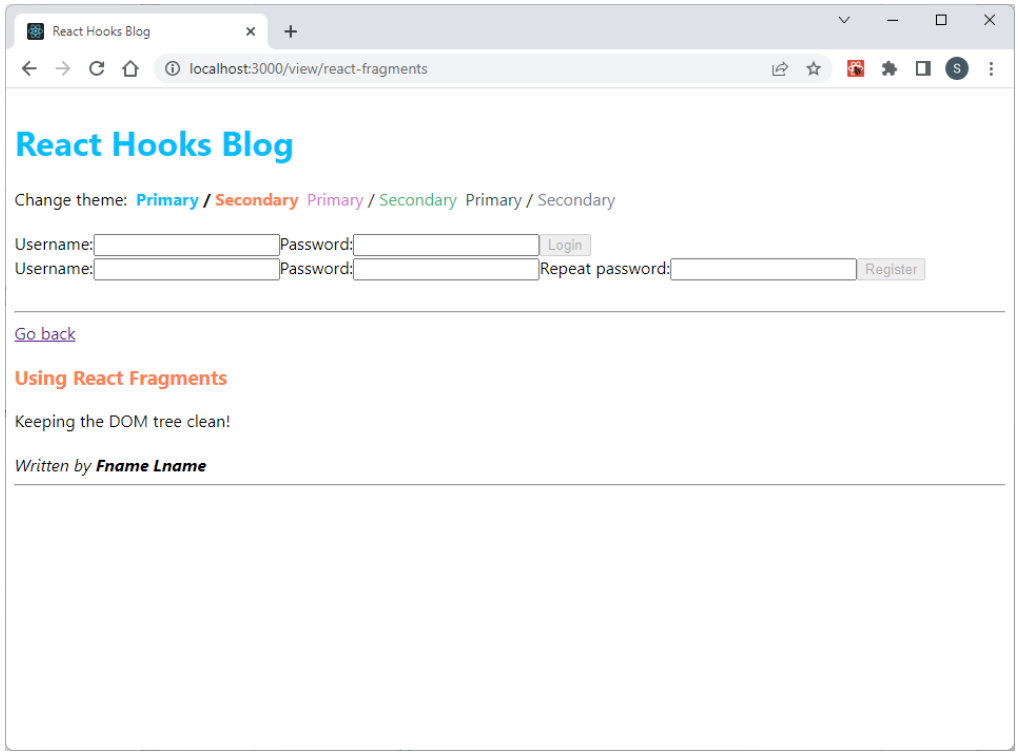
1. Edit `src/pages/PostPage.js`, and import the `Link` component there:

```
import { Link } from 'react-navi'
```

2. Then, insert a new link back to the main page, before displaying the post:

```
return (
    <div>
        <div><Link href="/">Go back</Link></div>
```

3. After going to a page, we can now use the **Go back** link in order to return to the main page:



Now, our app also provides a way back to the home page.

Step 4: Adjusting the CREATE_POST action

Previously, we dispatched a `CREATE_POST` action when a new post gets created. However, this action does not contain the post `id`, which means that links to newly created posts will not work.

We are now going to adjust the code to pass the post `id` to the `CREATE_POST` action:

1. Edit **`src/post/CreatePost.js`**, and import the `useEffect` Hook:

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

2. Next, adjust the existing Resource Hook to pull out the `post` object after the creation of the post finishes:

```
const [ post, createPost ] = useResource(({ title, content, author }) => ({
```

3. Now, we can create a new Effect Hook after the Resource Hook, and dispatch the `CREATE_POST` action once the result of the create post request becomes available:

```
useEffect(() => {
    if (post && post.data) {
        dispatch({ type: 'CREATE_POST', ...post.data })
    }
}, [post])
```

4. Next, we remove the call to the `dispatch` function in the `handleCreate` handler function:

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

5. Finally, we edit `src/reducers.js`, and adjust the `postsReducer` as follows:

```
function postsReducer (state, action) {
    switch (action.type) {
        case 'FETCH_POSTS':
            return action.posts

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

Now, links to the newly created posts work fine, because the `id` value is added to the inserted `post` object.

Exercise 3: Using routing Hooks(Chapter7_3)

After implementing basic routing using `navi` and `react-navi`, we are now going to implement more advanced use cases using routing Hooks, which are provided by `reactnavi`. Routing Hooks can be used to make routing more dynamic. For example, by allowing navigation to different routes from other Hooks. Furthermore, we can use Hooks to access all route-related information within a component.

Overview of Navi's Hooks

First, we will have a look at three of the Hooks that are provided by the Navi library:

- The `useNavigation` Hook
- The `useCurrentRoute` Hook
- The `useLoadingRoute` Hook

The useNavigation Hook

The `useNavigation` Hook has the following signature:

```
const navigation = useNavigation()
```

It returns the `navigation` object of Navi, which contains the following functions to manage the navigation state of the app:

- `extractState()`: Returns the current value of `window.history.state`; this is useful when dealing with server-side rendering.
- `getCurrentValue()` : Returns the `Route` object that corresponds to the current URL.
- `getRoute()`: Returns a promise to the fully loaded `Route` object that corresponds to the current URL. The promise will only resolve once the `Route` object is fully loaded.
- `goBack()`: Goes back one page; this is similar to how pressing the back button of the browser works.
- `navigate(url, options)`: Navigates to the provided URL using the provided options (`body`, `headers`, `method`, `replace`, and `state`). More information about the options can be found on the official Navi documentation:
  `https://frontarm.com/navi/en/reference/navigation/#navigationnavigate`

The useCurrentRoute Hook

The `useCurrentRoute` Hook has the following signature:

```
const route = useCurrentRoute()
```

It returns the latest non-busy route, which contains all information that Navi knows about the current page:

- `data`: Contains merged values from all `data` chunks.
- `title`: Contains the `title` value that should be set on `document.title`.
- `url`: Contains information about the current route, such as the `href`, `query`, and `hash`.
- `views`: Contains an array of components or elements that will be rendered in the route's view.

The useLoadingRoute Hook

The `useLoadingRoute` Hook has the following signature: `const`

```
loadingRoute = useLoadingRoute()
```

It returns the `Route` object for the page that is currently being fetched. If no page is currently being fetched, it outputs `undefined`. The object looks the same as the `Route` object of the `useCurrentRoute` Hook.

Step 1: Programmatic navigation

First, we are going to use the `useNavigation` Hook to implement programmatic navigation. We want to automatically redirect to the corresponding post page after creating a new post.

Let's implement programmatic navigation in the `CreatePost` component using Hooks:

1. Edit **`src/post/CreatePost.js`**, and import the `useNavigation` Hook there:

```
import { useNavigation } from 'react-navi'
```

2. Now, define a Navigation Hook after the existing Resource Hook:

```
const navigation = useNavigation()
```

3. Finally, we adjust the Effect Hook to call `navigation.navigate()`, once the result of the create post request becomes available:

```
useEffect(() => {
    if (post && post.data) {
        dispatch({ type: 'CREATE_POST', ...post.data })
        navigation.navigate(`/view/${post.data.id}`)
    }
}, [post])
```

If we create a new `post` object now, we can see that after pressing the **Create** button, we automatically get redirected to the page of the corresponding post. We can now move on to accessing route information using Hooks.

Step 2: Accessing route information

Next, we are going to use the `useCurrentRoute` Hook to access information about the current route/URL. We are going to use this Hook to implement a footer, which will display the `href` value of the current route.

Let's get started implementing the footer now:

1. First, we create a new component for the footer. Create a new **`src/pages/FooterBar.js`** file, and import `React`, as well as the `useCurrentRoute` Hook from `react-navi`:

```
import React from 'react'
import { useCurrentRoute } from 'react-navi'
```

2. Then, we define a new `FooterBar` component:

```
export default function FooterBar () {
```

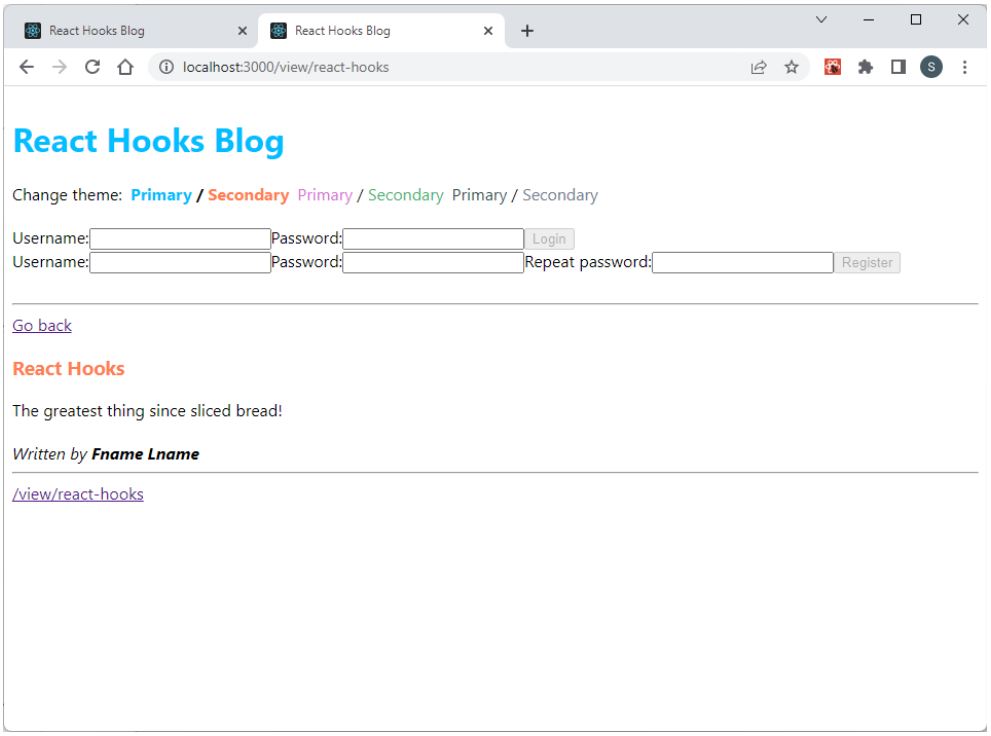
3. We use the `useCurrentRoute` Hook, and pull out the `url` object to be able to show the current `href` value in the footer:

```
    const { url } = useCurrentRoute()
```

4. Finally, we render a link to the current `href` value in the footer:

```
    return (
        <div>
            <a href={url.href}>{url.href}</a>
        </div>
    )
}
```

Now, when we, for example, open a post page, we can see the `href` value of the current post in the footer:



As we can see, our footer works properly—it always shows the `href` value of the current page.