

6 Implementing Requests and React Suspense

Exercise 1: Requesting resources with Hooks

In this section, we are going to learn how to request resources from a server, using Hooks. First, we are going to implement requests by only using the JavaScript `fetch` function, and the `useEffect/useState` Hooks. Then, we are going to learn how to request resources, using the `axios` library in combination with `react-request-hook`.

Step 1: Setting up a dummy server

Before we can implement requests, we need to create a backend server. Since we are focusing on the user interface at the moment, we are going to set up a dummy server, which will allow us to test out requests. We are going to use the `json-server` tool to create a full **Representational State Transfer (REST)** API from a JSON file.

Step 2: Creating the `db.json` file

To be able to use the `json-server` tool, first we need to create a `db.json` file, which is going to contain our full database for the server. The `json-server` tool will allow you to make the following:

- `GET` requests, to fetch data from the file
- `POST` requests, to insert new data into the file
- `PUT` and `PATCH` requests, to adjust existing data
- `DELETE` requests, to remove data

For all modifying actions (`POST`, `PUT`, `PATCH`, and `DELETE`), the updated file will automatically be saved by the tool.

We can use our existing structure for posts, which we defined as the default state of the posts reducer. However, we need to make sure that we provide an `id` value, so that we can query the database later:

```
[
  { "id": "react-hooks", "title": "React Hooks", "content": "The greatest
    thing since sliced bread!", "author": "Fname Lname" },
  { "id": "react-fragments", "title": "Using React Fragments", "content":
    "Keeping the DOM tree clean!", "author": "Fname Lname" }
]
```

As for the users, we need to come up with a way to store usernames and passwords. For simplicity, we just store the password in plain text (do not do this in a production environment!). Here, we also need to provide an `id` value:

```
[
  { "id": 1, "username": "Fname Lname", "password": "supersecure42" }
]
```

Additionally, we are going to store themes in our database. In order to investigate whether pulling themes from our database works properly, we are now going to define a third theme. As always, each theme needs an `id` value:

```
[
  { "id": 1, "primaryColor": "deepskyblue", "secondaryColor": "coral" },
  { "id": 2, "primaryColor": "orchid", "secondaryColor": "mediumseagreen"},
  { "id": 3, "primaryColor": "darkslategray", "secondaryColor": "slategray" }
]
```

Now, all that is left to do is to combine these three arrays into a single JSON object, by storing the posts array under a `posts` key, the users array under a `users` key, and the themes array under a `themes` key.

Let's start creating the JSON file that is used as a database for our backend server:

1. Create a new `server/` directory in the root of our application folder.
2. Create a `server/db.json` file with the following contents. We can use the existing state from our Reducer Hook. However, since this is a database, we need to give each element an `id` value (marked in bold):

```
{
  "posts": [
    { "id": "react-hooks", "title": "React Hooks", "content": "The
    greatest thing since sliced bread!", "author": "Fname Lname"
    },
    { "id": "react-fragments", "title": "Using React Fragments",
    "content": "Keeping the DOM tree clean!", "author":
    "Fname Lname" }
  ],
  "users": [
    { "id": 1, "username": "Fname Lname", "password":
    "supersecure42" }
  ],
  "themes": [
    { "id": 1, "primaryColor": "deepskyblue", "secondaryColor":
    "coral" },
    { "id": 2, "primaryColor": "orchid", "secondaryColor":
    "mediumseagreen" },
    { "id": 3, "primaryColor": "darkslategray",
    "secondaryColor": "slategray" }
  ]
}
```

For the `json-server` tool, we simply need a JSON file as the database, and the tool will create a full REST API for us.

Step 3: Installing the json-server tool

Now, we are going to install and start our backend server by using the json-server tool:

1. First, we are going to install the json-server tool via npm:

```
> npm install --save json-server
```

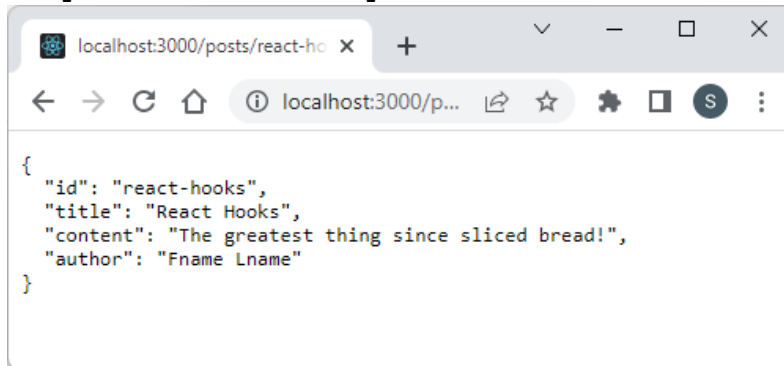
2. Now, we can start our backend server, by calling the following command:

```
> npx json-server --watch server/db.json
```

The npx command executes commands that were installed locally in a project. We need to use npx here, because we did not globally install the json-server tool (via `npm install -g json-server`).

We executed the json-server tool, and made it watch the server/db.json file that we created earlier. The --watch flag means that it will listen to changes to the file, and refresh automatically.

Now, we can go to <http://localhost:3000/posts/react-hooks> in order to see our post object:



As we can see, the tool created a full REST API from the database JSON file for us!

Step 4: Configuring package.json

Next, we need to adjust our `package.json` file, in order to start the server, in addition to our client (running via `webpack-dev-server`).

Let's start adjusting the `package.json` file:

1. First, we create a new package script called `start:server`, by inserting it in the `scripts` section of the `package.json` file. We also make sure that we change the port, so that it does not run on the same port as our client:

```
"scripts": {  
  "start:server": "npx json-server --watch server/db.json --port 4000",  
  "start": "react-scripts start",  
}
```

2. Then, we rename the `start` script to `start:client`:

```
"scripts": {  
  "start:server": "npx json-server --watch server/db.json",  
  "start:client": "react-scripts start",  
}
```

3. Next, we install a tool called `concurrently`, which lets us start the server and the client at the same time:

```
> npm install --save concurrently
```

4. Now, we can define a new `start` script by using the `concurrently` command, and then passing the server and client commands as arguments to it:

```
"scripts": {  
  "start": "npx concurrently \"npm run start:server\" \"npm run start:client\"",  
}
```

Now, running `npm start` will run the client, as well as the backend server.

Step 5: Configuring a proxy

Finally, we have to define a proxy, to make sure that we can request our API from the same **Uniform Resource Locator (URL)** as the client. This is needed because, otherwise, we would have to deal with cross-site requests, which are a bit more complicated to handle. We are going to define a proxy that will forward requests from `http://localhost:3000/api/` to `http://localhost:4000/`

Now, let's configure the proxy:

1. First, we have to install the `http-proxy-middleware` package:

```
> npm install --save http-proxy-middleware
```

2. Then, we create a new `src/setupProxy.js` file, with the following contents:

```
const proxy = require('http-proxy-middleware')

module.exports = function (app) {
  app.use(proxy('/api', {
```

3. Next, we have to define the target of our proxy, which will be the backend server, running at `http://localhost:4000`:

```
    target: 'http://localhost:4000',
```

4. Finally, we have to define a path-rewrite rule, which removes the `/api` prefix before forwarding the request to our server:

```
    pathRewrite: { '^/api': '' }
  }))
}
```

The preceding proxy configuration will link `/api` to our backend server; therefore, we can now start both the server and the client via the following command:

```
> npm start
```

Then, we can access the API by opening `http://localhost:3000/api/posts/react-hooks!`

Step 6: Defining routes

By default, the json-server tool defines the following routes: <https://github.com/typicode/json-server#routes>.

We can also define our own routes, by creating a `routes.json` file, where we can rewrite existing routes to other routes: <https://github.com/typicode/json-server#add-customroutes>.

For our blog app, we are going to define a single custom route:

`/login/:username/:password`. We are going to link this to a `/users?username=:username&password=:password` query, in order to find a user with the given username and password combination.

We are now going to define the custom login route for our app:

1. Create a new `server/routes.json` file with the following contents:

```
{
  "/login/:username/:password":
  "/users?username=:username&password=:password"
}
```

2. Then, adjust the `start:server` script in the `package.json` file, and add the `-routes` option, as follows:

```
"start:server": "npx json-server --watch server/db.json --port
4000 --routes server/routes.json",
```

3. From the command line, type and hit ENTER:

```
npm run start:server
```

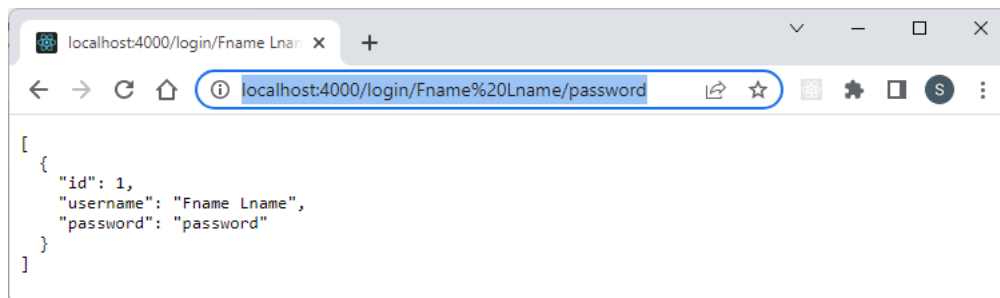
Now, our server will be serving our custom login route, which we are going to use later on in this lab!

4. We can try logging in by opening the following URL in our browser:

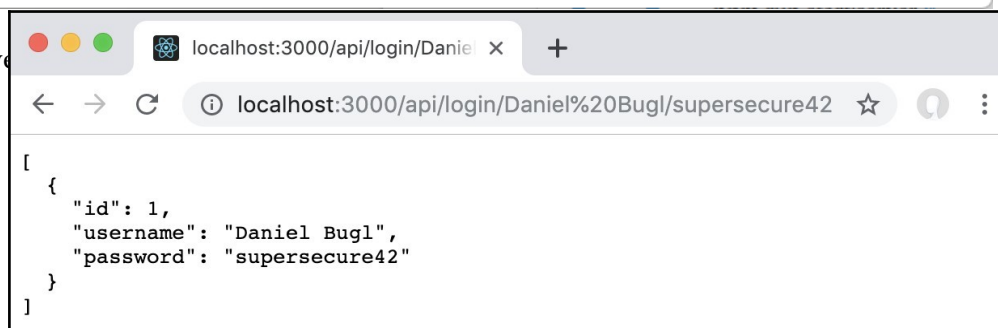
<http://localhost:4000/login/Fname%20Lname/password>.

This returns a user object; therefore, the login was successful!

We can see the user object being returned as text in our browser:



As we



Exercise 2: Implementing requests using Effect and State/Reducer Hooks(Chapter6_2)

Before we use a library to implement requests using Hooks, we are going to implement them manually, using an Effect Hook to trigger the request, and State/Reducer Hooks to store the result.

Requests with Effect and State Hooks

First, we are going to request themes from our server, instead of hardcoding the list of themes.

Step 1: Let's implement requesting themes using an Effect Hook and a State Hook:

1. In the **src/ChangeTheme.js** file, adjust the `React` import statement in order to import the `useEffect` and `useState` Hooks:

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

2. Remove the `THEMES` constant, which is all of the following code:

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

3. In the `ChangeTheme` component, define a new `useState` Hook in order to store the themes:

```
export default function ChangeTheme ({ theme, setTheme }) {  
  const [ themes, setThemes ] = useState([])
```

4. Then define a `useEffect` Hook, where we are going to make the request:

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

5. In this Hook, we use `fetch` to request a resource; in this case, we request `/api/themes`:

```
  fetch('/api/themes')
```


6. Fetch makes use of the Promise API; therefore, we can use `.then()` in order to work with the result. First, we have to parse the result as JSON:

```
.then(result => result.json())
```

7. Finally, we call `setThemes` with the `themes` array from our request:

```
.then(themes => setThemes(themes))
```

We can also shorten the preceding function to `.then(setThemes)`, as we are only passing down the `themes` argument from `.then()`.

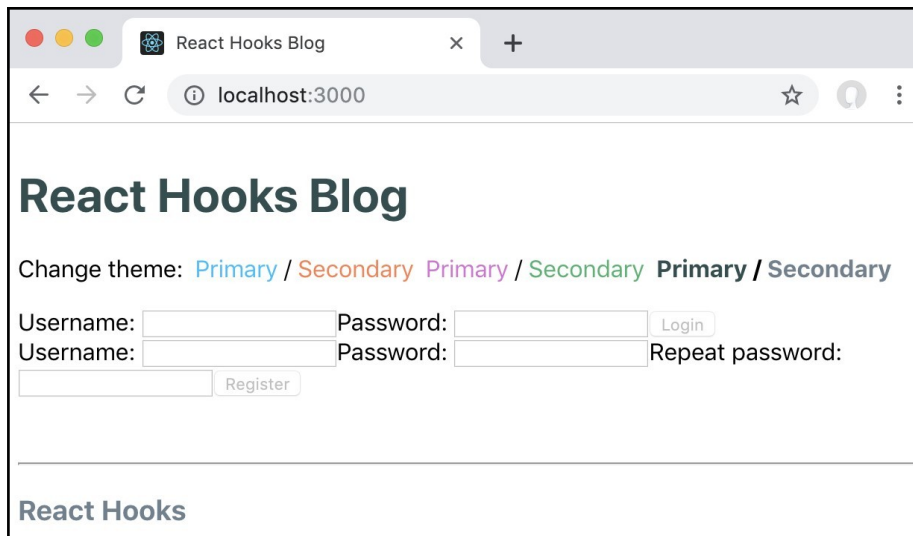
8. For now, this Effect Hook should only trigger when the component mounts, so we pass an empty array as the second argument to `useEffect`. This ensures that the Effect Hook has no dependencies, and thus will only trigger when the component mounts:

```
}, [])
```

9. Now, all that is left to do is to replace the `THEMES` constant with our `themes` value from the Hook:

```
{ themes.map(t =>
```

As we can see, there are now three themes available, all loaded from our database through our server:



Three themes loaded from our server by using hooks!

Our themes are now loaded from the backend server and we can move on to requesting posts via Hooks.

Requests with Effect and Reducer Hooks

We are now going to use our backend server to request the `posts` array, instead of hardcoding it as the default value for the `postsReducer`.

Step 2: Let's implement requesting posts using an Effect Hook and a Reducer Hook:

1. **Remove** the `defaultPosts` constant definition from `src/App.js`, which is all of the following code:

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

2. Replace the `defaultPosts` constant in the `useReducer` function with an empty array:

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

3. In `src/reducers.js`, define a new action type, called `FETCH_POSTS`, in the `postsReducer` function. This action type will replace the current state with a new posts array:

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

4. In `src/App.js`, define a new `useEffect` Hook, which precedes the current one:

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

5. In this Hook, we again use `fetch` in order to request a resource; in this case, we request `/api/posts`:

```
  fetch('/api/posts')
    .then(result => result.json())
```

6. Finally, we dispatch a `FETCH_POSTS` action with the `posts` array from our request:

```
    .then(posts => dispatch({ type: 'FETCH_POSTS', posts
}))
```

7. For now, this Effect Hook should only trigger when the component mounts, so we pass an empty array as the second argument to `useEffect`:

```
  }, [])
```

As we can see, the posts now get requested from the server! We can have a look at the DevTools **Network** tab to see the request:

The screenshot displays a web browser with two pages. The left page, titled "React Hooks", features the text "The greatest thing since sliced bread!" and "Written by **Daniel Bugl**". The right page, titled "Using React Fragments", features the text "Keeping the DOM tree clean!" and "Written by **Daniel Bugl**". A Chrome DevTools Network tab is open on the right, showing a list of requests. The selected request is a GET request to "http://localhost:3000/api/fragments", which returned a 200 status and a JSON response containing two fragment objects.

Posts being requested from our server!

The posts are now being requested from the backend server. In the next section, we are going to use `axios` and the `react-request-hook` to request resources from our server.

Exercise 3: Using axios and react-request-hook (Chapter6_2)

In the previous section, we used an Effect Hook to trigger the request, and a Reducer/State Hook to update the state, using the result from the request. Instead of manually implementing requests like this, we can use the `axios` and `react-request-hook` libraries to easily implement requests using Hooks.

Setting up the libraries

Before we can start using `axios` and `react-request-hook`, we have to set up an `axios` instance and a `RequestProvider` component.

Step 1: Let's get started setting up the libraries:

1. First, we install the libraries:

```
> npm install --save react-request-hook axios
```

2. Then, we import them in `src/index.js`:

```
import { RequestProvider } from 'react-request-hook'
import axios from 'axios'
```

3. Now, we define an `axios` instance, where we set the `baseUrl` to `http://localhost:3000/api/`—our backend server:

```
const axiosInstance = axios.create({
  baseUrl: 'http://localhost:3000/api/'
})
```

In the config for our `axios` instance, we can also define other options, such as a default timeout for requests, or custom headers. For more information, check out the [axios documentation](https://github.com/axios/axios#axioscreateconfig):

<https://github.com/axios/axios#axioscreateconfig>.

4. Finally, we wrap our `<App />` component with the `<RequestProvider>` component. Remove the following line of code:

```
ReactDOM.render(<App />, document.getElementById('root'));
```

Replace it with this code:

```
ReactDOM.render(
  <RequestProvider value={axiosInstance}>
    <App />
  </RequestProvider>,
  document.getElementById('root') )
```

Now, our app is ready to use Resource Hooks!

Using the useResource Hook

A more powerful way of dealing with requests, is using the `axios` and `react-requesthook` libraries. Using these libraries, we have access to features that can cancel a single request, or even clear all pending requests. Furthermore, using these libraries makes it easier to deal with errors and loading states.

Step 2: We are now going to implement the `useResource` Hook in order to request themes from our server:

1. In `src/ChangeTheme.js`, import the `useResource` Hook from the `react-request-hook` library:

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

2. Remove the previously defined State and Effect Hooks.
3. Then, we define a `useResource` Hook within the `ChangeTheme` component. The Hook returns a value and a getter function. Calling the getter function will request the resource:

```
export default function ChangeTheme ({ theme, setTheme }) {  
  const [ themes, getThemes ] = useResource(() => ({
```

Here, we are using the shorthand syntax for `() => { return { } }`, which is `() => ({ })`. Using this shorthand syntax allows us to concisely write functions that only return an object.

4. In this Hook we pass a function, which returns an object with information about the request:

```
    url: '/themes',  
    method: 'get'  
  )))
```

With `axios`, we only need to pass `/themes` as the `url`, because we already defined the `baseUrl`, which contains `/api/`.

5. The Resource Hook returns an object with a `data` value, an `isLoading` boolean, an error object, and a cancel function to cancel the pending request. Now, we pull out the `data` value and the `isLoading` boolean from the `themes` object:

```
const { data, isLoading } = themes
```

6. Then, we define a `useEffect` Hook to trigger the `getThemes` function. We only want it to trigger once, when the component mounts; therefore, we pass an empty array as the second argument:

```
useEffect(getThemes, [])
```

7. Additionally, we use the `isLoading` flag to display a loading message while waiting for the server to respond:

```
{isLoading && ' Loading themes...'}
```

8. Finally, we rename the `themes` value to the `data` value that is returned from the `useResource` Hook, and add a conditional check to ensure the `data` value is already available:

```
{data && data.map(t =>
```

If we have a look at our app now, we can see that the **Loading themes...** message gets displayed for a very short time, and, then the themes from our database get displayed! We can now move on to requesting posts using the Resource Hook.

Using useResource with a Reducer Hook

The `useResource` Hook already handles the state for the result of our request, so we do not need an additional `useState` Hook to store the state. If we already have an existing Reducer Hook, however, we can use it in combination with the `useResource` Hook.

Step 3: We are now going to implement the `useResource` Hook in combination with a Reducer Hook in our app:

1. In `src/App.js`, import the `useResource` Hook from the `react-request-hook` library:

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

2. Remove the previously defined `useEffect` Hook that uses `fetch` to request `/api/posts`.

3. Define a new `useResource` Hook, where we request `/posts`:

```
const [ posts, getPosts ] = useResource(() => ({
  url: '/posts',
  method: 'get'
}))
```

4. Define a new `useEffect` Hook, which simply calls `getPosts`:

```
useEffect(getPosts, [])
```

5. Finally, define a `useEffect` Hook, which dispatches the `FETCH_POSTS` action, after checking if the data already exists:

```
useEffect(() => {
  if (posts && posts.data) {
    dispatch({ type: 'FETCH_POSTS', posts: posts.data })
  }
})
```

6. We make sure that this Effect Hook triggers every time the `posts` object updates:

```
}, [posts])
```

Now, when we fetch new posts, a `FETCH_POSTS` action will be dispatched. Next, we move on to handling errors during requests.

Handling error state

We have already handled the loading state in the `ChangeTheme` component. Now, we are going to implement the error state for posts.

Step 4: Let's get started handling the error state for posts:

1. In `src/reducers.js`, define a new `errorReducer` function with a new action type, `POSTS_ERROR`:

```
function errorReducer (state, action) {
  switch (action.type) {
    case 'POSTS_ERROR':
      return 'Failed to fetch posts'
    default:
      return state
  }
}
```

```
}
```

2. Add the `errorReducer` function to our `appReducer` function:

```
export default function appReducer (state, action) {  
  return {  
    user: userReducer(state.user, action),  
    posts: postsReducer(state.posts, action),  
    error: errorReducer(state.error, action)  
  }  
}
```

3. In `src/App.js`, adjust the default state of our Reducer Hook:

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

4. Pull the error value out of the state object:

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

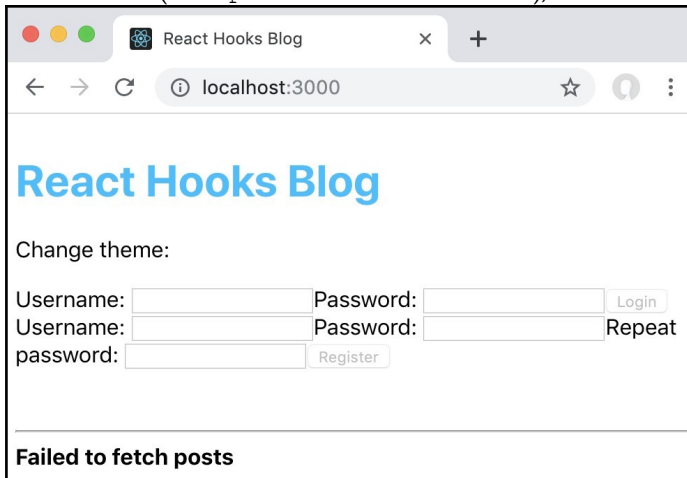
5. Now, we can adjust the existing Effect Hook that handles new data from the `posts` resource, by dispatching a `POSTS_ERROR` action in the case of an error:

```
useEffect(() => {  
  if (posts && posts.error) {  
    dispatch({ type: 'POSTS_ERROR' })  
  }  
  if (posts && posts.data) {  
    dispatch({ type: 'FETCH_POSTS', posts: posts.data })  
  }  
}, [posts])
```

6. Finally, we display the error message before the `PostList` component:

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

If we only start the client now (via `npm run start:client`), the error will be displayed:



Displaying an error when the request fails!

As we can see, the **Failed to fetch posts** error gets displayed in our app, because the server is not running. We can now move on to implementing post creation via requests.

Implementing post creation

Now that we have a good grasp on how to request data from an API, we are going to use the `useResource` Hook for the creation of new data.

Step 5: Let's get started implementing post creation using the Resource Hook:

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

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

2. Then, define a new Resource Hook, below the other Hooks, but before our handler function definitions. Here, we set the method to `post` (creates new data) and we pass the data from the `createPost` function, to the request config:

```
const [ , createPost ] = useResource(({ title, content, author  
}) => ({  
  url: '/posts',  
  method: 'post',  
  data: { title, content, author }  
}))
```

Here, we are using a shorthand syntax for array destructuring: we are ignoring the first element of the array, by not specifying a value name. Instead of writing `const [post, createPost]`, and then not using `post`, we just put a comma, as follows: `const [, createPost]`.

3. Now, we can use the `createPost` function in our `handleCreate` handler function. We make sure that we keep the call to the `dispatch` function there, so that we immediately insert the new post client-side, while waiting for the server to respond. The added code is highlighted in bold:

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

Please note that, in this simple example, we do not expect, or handle the failure of post creations. In this case, we dispatch the action even before the request completes. However, when implementing login, we are going to handle error states from the request, in order to check whether the user was logged in successfully. It is best practice to always handle error states in real-world applications.

4. Note that when we insert a post now, the post will first be at the beginning of the list; however, after refreshing, it will be at the end of the list. Unfortunately, our server inserts new posts at the end of the list. Therefore, we are going to reverse the order, after fetching posts from the server. Edit `src/App.js`, and adjust the following code:

```
if (posts && posts.data) {  
  dispatch({ type: 'FETCH_POSTS',  
    posts: posts.data.reverse() })  
}
```

Now, inserting a new post via the server works fine and we can move on to implementing registration!

Implementing registration

Next, we are going to implement registration, which is going to work in very similar way to creating posts.

Step 6: Let's get started implementing registration:

1. First, import the `useEffect` and `useResource` Hooks in `src/user/Register.js`:

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

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

2. Then, define a new `useResource` Hook, below the other Hooks, and before the handler functions. Unlike we did in the post creation, we now want to also store the resulting user object:

```
const [ user, register ] = useResource((username, password) =>
({
  url: '/users',
  method: 'post',
  data: { username, password }
}))
```

3. Next, define a new `useEffect` Hook below the `useResource` Hook, which will dispatch a REGISTER action when the request completes:

```
useEffect(() => {
  if (user && user.data) {
    dispatch({ type: 'REGISTER', username: user.data.username })
  }
}, [user])
```

Please note that, in this simple example, we do not expect, or handle the failure of registrations. In this case, we dispatch the action only after the successful creation of the user. However, when implementing login, we are going to handle error states from the request, in order to check whether the user was logged in successfully. It is best practice to always handle error states in real-world applications.

4. Finally, we adjust the form submit handler in order to call the `register` function, instead of directly dispatching the action:

```
<form onSubmit={e => { e.preventDefault();
  register(username, password) }}>
```

Now, if we enter a **Username** and **Password**, and press **Register**, a new user will be inserted into our `db.json` file and, just like before, we will be logged in. We now move on to implementing login via Resource Hooks.

Implementing login

Finally, we are going to implement login, via requests using our custom route. After doing so, our blog app will be fully connected to the server.

Step 7: Let's get started implementing login:

1. First, edit **src/user/Login.js** and import the **useEffect** and **useResource** Hooks:

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

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

2. We define a new State Hook that will store a boolean to check if the login failed:

```
const [ loginFailed, setLoginFailed ] = useState(false)
```

3. Then, we define a new State Hook for the **Password** field, because we did not handle it before:

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

4. Now, we define a handler function for the **Password** field, below the **handleUsername** function:

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

5. Next, we handle the value change in the input field:

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

6. Now, we can define our Resource Hook below the State Hooks, where we are going to pass username and password to the /login route. Since we are passing them as part of the URL, we need to make sure that we encode them properly first:

```
const [ user, login ] = useResource((username, password) => ({
  url:
`/login/${encodeURIComponent(username)}/${encodeURIComponent(password)}`,
  method: 'get'
}))
```

Please note that it is not secure to send the password in cleartext via a GET request. We only do this for the sake of simplicity when configuring our dummy server. In a real world application, use a POST request for login instead and send the password as part of the POST data. Also make sure to use **Hypertext Transfer Protocol Secure (HTTPS)** so that the POST data will be encrypted.

7. Next, we define an Effect Hook, which will dispatch the LOGIN action if the request completes successfully:

```
useEffect(() => {
  if (user && user.data) {
```

8. Because the login route returns either an empty array (login failed), or an array with a single user, we need to check whether the array contains at least one element:

```
    if (user.data.length > 0) {
      setLoginFailed(false)
      dispatch({ type: 'LOGIN', username:
        user.data[0].username })
    } else {
```

9. If the array was empty, we set loginFailed to true:

```
      setLoginFailed(true)
    }
  }
```

10. If we get an error response from the server, we also set the login state to failed:

```
if (user && user.error) {  
  setLoginFailed(true)  
}
```

11. We make sure that the Effect Hook triggers whenever the `user` object from the Resource Hook updates:

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

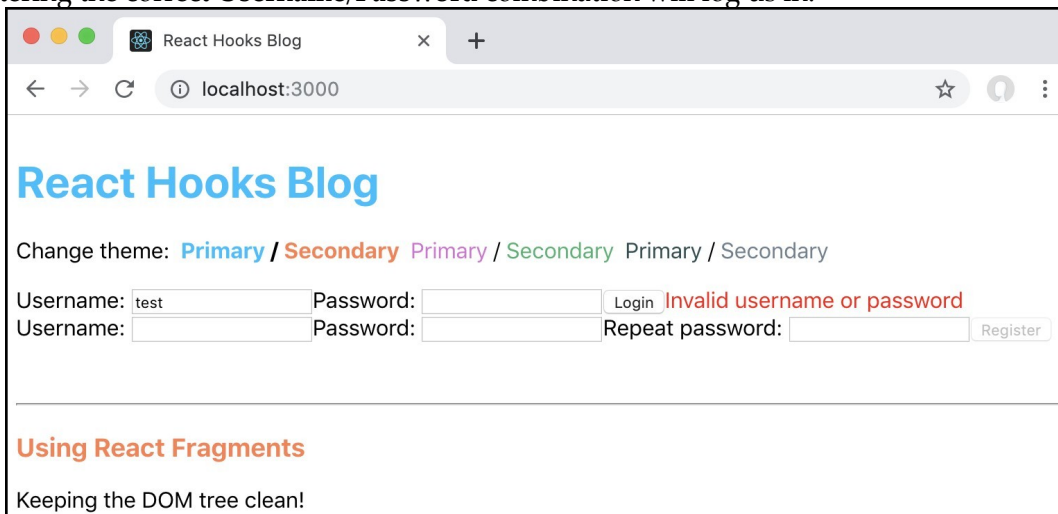
12. Then, we adjust the `onSubmit` function of form, in order to call the `login` function:

```
<form onSubmit={e => { e.preventDefault(); login(username,  
password) }}>
```

13. Finally, below the **Submit** button, we display the **Invalid username or password** message, in case `loginFailed` was set to `true`:

```
{loginFailed && <span style={{ color: 'red' }}>Invalid username or  
password</span>}
```

As we can see, entering a wrong **Username** or **Password** (or no **Password**) will result in an error, while entering the correct **Username/Password** combination will log us in:



Displaying an error message when the login failed

Now, our app is fully connected to the backend server!

Exercise 2: Preventing unnecessary re-rendering with `React.memo`

With class components we had `shouldComponentUpdate`, which would prevent components from re-rendering if the props did not change.

With function components, we can do the same using `React.memo`, which is a higher-order component. `React.memo` memoizes the result, which means that it will remember the last rendered result, and, in cases where the props did not change, it will skip re-rendering the component:

```
const SomeComponent = () => ...

export default React.memo(SomeComponent)
```

By default, `React.memo` will act like the default definition of `shouldComponentUpdate`, and it will only shallowly compare the props object. If we want to do a special comparison, we can pass a function as a second argument to `React.memo`:

```
export default React.memo(SomeComponent, (prevProps, nextProps) => {
  // compare props and return true if the props are equal and we should
  not update
})
```

Unlike `shouldComponentUpdate`, the function that is passed to `React.memo` returns `true` when the props are equal, and thus it should not update, which is the opposite of how `shouldComponentUpdate` works! After learning about `React.memo`, let's try it out in practice by implementing `React.memo` for the `Post` component.

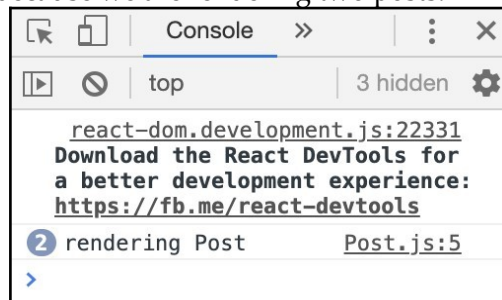
Implementing React.memo for the Post component (Chapter6_4)

Step 1: First, let's find out when the `Post` component re-renders. To do this, we are going to add a `console.log` statement to our `Post` component, as follows:

1. Edit `src/post/Post.js`, and add the following debug output when the component renders:

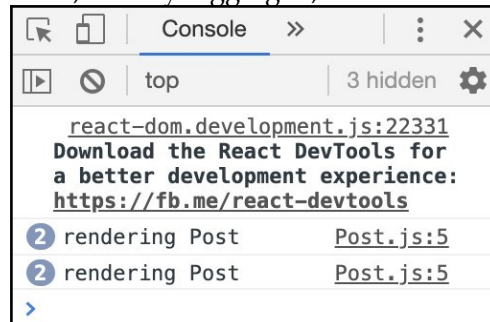
```
export default function Post ({ title, content, author }) {  
  console.log('rendering Post')
```

2. Now, open the app at `http://localhost:3000`, and open the DevTools (on most browsers: right-click | **Inspect** on the page). Go to the **Console** tab, and you should see the output twice, because we are rendering two posts:



The debug output when rendering two posts

3. So far, so good. Now, let's try logging in, and see what happens:



Posts re-rendering after logging in

As we can see, the `Post` components unnecessarily re-render after logging in, although their props did not change. We can use `React.memo` to prevent this, as follows:

1. Edit `src/post/Post.js`, and remove the `export default` part of the function definition (marked in bold):

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

2. Then, at the bottom of the file, export the `Post` component, after wrapping it with `React.memo()`:

```
export default React.memo(Post)
```

3. Now, refresh the page and log in again. We can see that the two posts get rendered, which produces the initial debug output. However, logging in now does not cause the Post components to re-render anymore!

If we wanted to do a custom check on whether the posts are equal, we could, for example, compare title, content, and author, as follows:

```
export default React.memo(Post,  
  (prev, next) => prev.title === next.title && prev.content ===  
  next.content && prev.author === next.author  
)
```

In our case, doing this will have the same effect, because React already does a shallow comparison of all props, by default. This function only becomes useful when we have deep objects to compare, or when we want to ignore changes in certain props. Please note that we should not prematurely optimize our code. Re-renders can be fine, since React is intelligent, and does not paint to the browser if nothing changed. Therefore, it might be overkill to optimize all re-renders, unless a certain case has already been identified as a performance bottleneck.

Exercise 3: Implementing lazy loading with React Suspense(Chapter6_5)

React Suspense allows us to let components wait before rendering. At the moment, React Suspense only allows us to dynamically load components with `React.lazy`. In the future, Suspense will support other use cases, such as data fetching.

`React.lazy` is another form of performance optimization. It lets us load a component dynamically in order to reduce the bundle size. Sometimes we want to avoid loading all of the components during the initial render, and only request certain components when they are needed.

For example, if our blog has a member area, we only need to load it after the user has logged in. Doing this will reduce the bundle size for guests who only visit our blog to read blog posts. To learn about React Suspense, we are going to lazily load the `Logout` component in our blog app.

Step 1: Implementing React.Suspense

First, we have to specify a loading indicator, which will be shown when our lazily-loaded component is loading. In our example, we are going to wrap the `UserBar` component with React Suspense.

Edit `src/App.js`, and replace the `<UserBar />` component with the following code:

```
<React.Suspense fallback={"Loading..."}>
  <UserBar />
</React.Suspense>
```

Now, our app is ready for implementing lazy loading.

Step 2: Implementing React.lazy

Next, we are going to implement lazy loading for the `Logout` component by wrapping it with `React.lazy()`, as follows:

1. Edit `src/user/UserBar.js`, and remove the import statement for the `Logout` component:

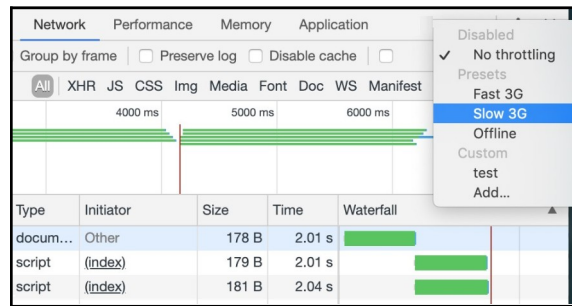
```
import Logout from './Logout'
```

2. Then, define the `Logout` component via lazy loading:

```
const Logout = React.lazy(() => import('./Logout'))
```

The `import()` function dynamically loads the `Logout` component from the `Logout.js` file. In contrast to the static `import` statement, this function only gets called when `React.lazy` triggers it, which means it will only be imported when the component is needed.

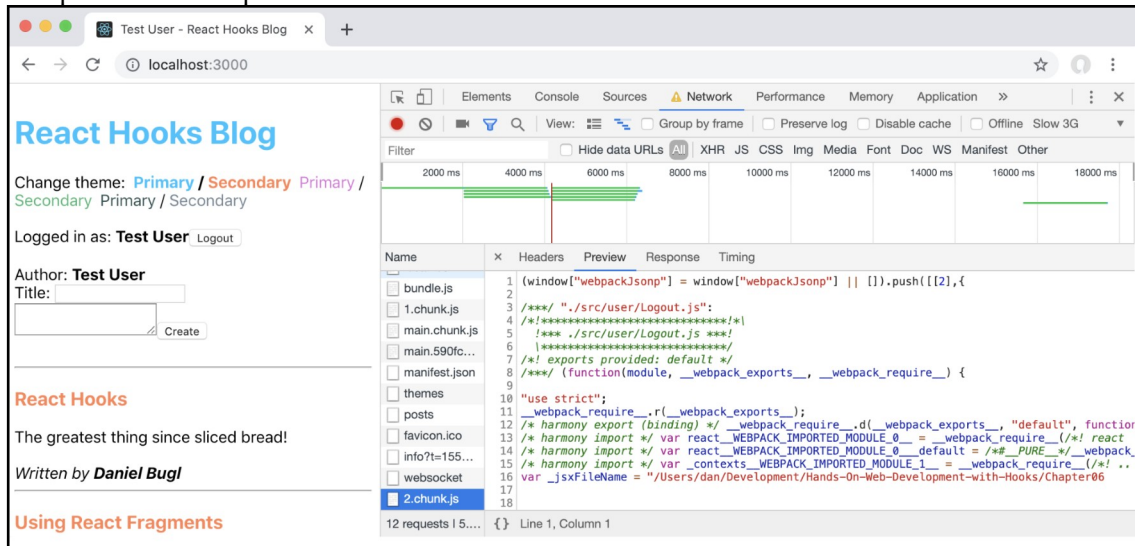
If we want to see lazy loading in action, we can set **Network Throttling** to **Slow 3G** in Google Chrome:



Setting Network Throttling to Slow 3G in Google Chrome

In Firefox, we can do the same by setting **Network Throttling** to **GPRS**. Safari unfortunately does not offer such a feature right now, but we can use the **Network Link Conditioner** tool from Apple's "Hardware IO tools": <https://developer.apple.com/download/more/>

If we refresh the page now, and then log in, we can first see the **Loading...** message, and then the Logout component will be shown. If we take a look at the **Network** logs, we can see that the Logout component was requested via the network:



The screenshot shows a web browser at localhost:3000 displaying the 'React Hooks Blog'. The page includes a theme selector, a login status 'Logged in as: Test User' with a 'Logout' button, and a form for creating a post. The network tab is open, showing a list of files. The file '2.chunk.js' is selected, and its source code is visible in the preview pane. The source code is a JavaScript module that defines a 'Logout' component and exports it.

```
1 (window["webpackJsonp"] = window["webpackJsonp"] || []).push([[2], {
2
3   /**/ " ./src/user/Logout.js":
4   /**/ {
5     /**/ " ./src/user/Logout.js ***!
6     /**/ {
7       /**/ exports provided: default */
8       /**/ {function(module, __webpack_exports__, __webpack_require__) {
9
10        "use strict";
11        __webpack_require__.r(__webpack_exports__);
12        /** harmony export (binding) */ __webpack_require__.d(__webpack_exports__, "default", function() {
13          /** harmony import */ var react__WEBPACK_IMPORTED_MODULE_0___ = __webpack_require__(/*! react */
14          /** harmony import */ var react__WEBPACK_IMPORTED_MODULE_0___ = __webpack_require__(/*! react */
15          /** harmony import */ var react__WEBPACK_IMPORTED_MODULE_0___ = __webpack_require__(/*! react */
16          var _jsxFileName = "/Users/dan/Development/Hands-On-Web-Development-with-Hooks/Chapter06
17
18
```

The Logout component being loaded via the network

As we can see, the Logout component is now lazily loaded, which means that it will only be requested when needed.