# 8 Using Community Hooks

## Exercise 1: Exploring the input handling Hook

A very common use case when dealing with Hooks, is to store the current value of an `input` field using State and Effect Hooks.

The `useInput` Hook greatly simplifies this use case, by providing a single Hook that deals with the `value` variable of an `input` field. It works as follows:

```
import React from 'react'
import { useInput } from 'react-hookedup'

export default function App () {

    const { value, onChange } = useInput('')

    return <input value={value} onChange={onChange} />
}
```

This code will bind an `onChange` handler function and `value` to the `input` field. This means that whenever we enter text into the `input` field, the `value` will automatically be updated.

Additionally, there is a function that will clear the `input` field. This `clear` function is also returned from the Hook:

```
const { clear } = useInput('')
```

Calling the `clear` function will set the `value` to an empty value, and clear all text from the `input` field.
Furthermore, the Hook provides two ways to bind an `input` field:

- `bindToInput`: Binds the `value` and `onChange` props to an `input` field using `e.target.value` as the `value` argument for the `onChange` function. This is useful when dealing with HTML `input` fields.
- `bind`: Binds the `value` and `onChange` props to an `input` field using only `e` as the value for the `onChange` function. This is useful for React components that directly pass the value to the `onChange` function.

The `bind` and `bindToInput` objects can be used with the spread operator, as follows:

```
import React from 'react'
import { useInput } from 'react-hookedup'

const ToggleButton = ({ value, onChange }) => { ... } // custom component that renders
a toggle button

export default function App () {
    const { bind, bindToInput } = useInput('')

    return (
        <div>
            <input {...bindToInput} />
            <ToggleButton {...bind} />
        </div>
    )
}
```

As we can see, for the `input` field we can use the `{...bindToInput}` props to assign the `value` and `onChange` functions. For `ToggleButton`, we need to use the `{...bind}` props instead, because we are not dealing with input events here, and the value is directly passed to the change handler (not via `e.target.value`).

Now that we have learned about the Input Hook, we can move on to implementing it in our blog app.

# Exercise 2: Implementing Input Hooks in our blog app

Now that we have learned about the Input Hook, and how it simplifies dealing with the `input` field state, we are going to implement Input Hooks in our blog app.

First, we have to install the `react-hookedup` library in our blog app project:

```
> npm install --save react-hookedup
```

We are now going to implement Input Hooks in the following components:

- The `Login` component
- The `Register` component
- The `CreatePost` component

Let's get started implementing Input Hooks.

## Step 1: The Login component

We have two `input` fields in the `Login` component: the **Username** and **Password** fields. We are now going to replace the State Hooks with Input Hooks.

Let's start implementing Input Hooks in the `Login` component now:

1. Import the `useInput` Hook at the beginning of the **`src/user/Login.js`** file:

```
import { useInput } from 'react-hookedup'
```

2. Then, we remove the following `username` State Hook:

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

   It is replaced with an Input Hook, as follows:

```
const { value: username, bindToInput: bindUsername } = useInput('')
```

   Since we are using two Input Hooks, in order to avoid name collisions, we are using the rename syntax (`{ from: to }`) in object destructuring to rename the `value` key to `username`, and `bindToInput` key to `bindUsername`.

3. We also remove the following `password` State Hook:

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

   It is replaced with an Input Hook, as follows:

```
const { value: password, bindToInput: bindPassword } = useInput('')
```

4. We can now remove the following handler functions:

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

5. Finally, instead of passing the `onChange` handlers manually, we use the bind objects from the Input Hooks:

```
            <input type="text" value={username} {...bindUsername}
name="login-username" id="login-username" />
            <input type="password" value={password}
{...bindPassword} name="login-password" id="login-password" />
```

The login functionality will still work in exactly the same way as before, but we are now using the much more concise Input Hook, instead of the generic State Hook. We also do not have to define the same kind of handler function for each `input` field anymore. As we can see, using community Hooks can greatly simplify the implementation of common usecases, such as input handling. We are now going to repeat the same process for the `Register` component.

## Step 2: The Register component

The `Register` component works similarly to the `Login` component. However, it has three `input` fields: **Username**, **Password**, and **Repeat Password**.

Let's implement Input Hooks in the `Register` component now:

1. Import the `useInput` Hook at the beginning of the **`src/user/Register.js`** file:

```
import { useInput } from 'react-hookedup'
```

2. Then, we remove the following State Hooks:

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

   They are replaced with the corresponding Input Hooks:

```
    const { value: username, bindToInput: bindUsername } =
useInput('')
    const { value: password, bindToInput: bindPassword } =
useInput('')
    const { value: passwordRepeat, bindToInput: bindPasswordRepeat } =
useInput('')
```

3. Again, we can remove all of the handler functions:

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

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

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

4. Finally, we replace all of the `onChange` handlers with the corresponding bind objects:

```
            <input type="text" value={username} {...bindUsername}
name="register-username" id="register-username" />
<input type="password" value={password}
{...bindPassword} name="register-password" id="register-password"
/>
            <input type="password" value={passwordRepeat}
{...bindPasswordRepeat} name="register-password-repeat" id="register-
password-repeat/>
```

The register functionality will also still work in the same way, but now using Input Hooks. Next up is the `CreatePost` component, where we are going to implement Input Hooks as well.

## Step 3: The CreatePost component

The `CreatePost` component uses two `input` fields: one for the `title`, and one for the `content`. We are going to replace both of them with Input Hooks.
Let's implement Input Hooks in the `CreatePost` component now:

1. Import the `useInput` Hook at the beginning of the **`src/post/CreatePost.js`** file:

```
import { useInput } from 'react-hookedup'
```

2. Then, we remove the following State Hooks:

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

   We replace them with the corresponding Input Hooks:

```
const { value: title, bindToInput: bindTitle } = useInput('')
const { value: content, bindToInput: bindContent } = useInput('')
```

3. Again, we can remove the following input handler functions:

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

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

4. Finally, we replace all of the `onChange` handlers with the corresponding bind objects:

```
                <input type="text" value={title} {...bindTitle}
name="create-title" id="create-title" />
        </div>
        <textarea value={content} {...bindContent} />
```

The create post functionality will also work in the same way with Input Hooks.

# Exercise 3: React life cycles with Hooks (Chapter8_2)

As we have learned in the previous labs, we can use the `useEffect` Hook to model most of React's life cycle methods. However, if you prefer dealing with React life cycle directly, instead of using Effect Hooks, there is a library called `react-hookedup`, which provides various Hooks, including Hooks for the various React life cycles. Additionally, the library provides a merging State Hook, which works similarly to `this.setState()` in React's class components.

## Step 1: The useOnMount Hook

The `useOnMount` Hook has a similar effect to the `componentDidMount` life cycle. It is used as follows:

```
import React from 'react'
import { useOnMount } from 'react-hookedup'

export default function UseOnMount () {
    useOnMount(() => console.log('mounted'))

    return <div>look at the console :)</div> }
```

The preceding code will output **mounted** to the console when the component gets mounted (when the React component is rendered for the first time). It will not be called again when the component re-renders due to, for example, a prop change.

Alternatively, we could just use a `useEffect` Hook with an empty array as the second argument, which will have the same effect:

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

export default function OnMountWithEffect () {     useEffect(() =>
console.log('mounted with effect'), [])

    return <div>look at the console :)</div> }
```

As we can see, using an Effect Hook with an empty array as the second argument results in the same behavior as the `useOnMount` Hook or the `componentDidMount` life cycle method.

## Step 2: The useOnUnmount Hook

The `useOnUnmount` Hook has a similar effect to the `componentWillUnmount` life cycle. It is used as follows:

```
import React from 'react'
import { useOnUnmount } from 'react-hookedup'

export default function UseOnUnmount () {
    useOnUnmount(() => console.log('unmounting'))
    return <div>click the "unmount" button above and look at the console </div> }
```

The preceding code will output **unmounting** to the console when the component gets unmounted (before the React component is removed from the DOM).

If you remember from `Chapter 4`, *Using the Reducer and Effect Hooks*, we can return a cleanup function from the `useEffect` Hook, which will be called when the component unmounts. This means that we could alternatively implement the `useOnMount` Hook using `useEffect`, as follows:

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

export default function OnUnmountWithEffect () {
    useEffect(() => {
        return () => console.log('unmounting with effect')
    }, [])

    return <div>click the "unmount" button above and look at the console</div> }
```

As we can see, using the cleanup function that is returned from an Effect Hook, with an empty array as the second argument, has the same effect as the `useOnUnmount` Hook, or the `componentWillUnmount` life cycle method.

## Step 3: The useLifecycleHooks Hook

The `useLifecycleHooks` Hook combines the previous two Hooks into one. We can combine the `useOnMount` and `useOnUnmount` Hooks as follows:

```
import React from 'react'
import { useLifecycleHooks } from 'react-hookedup'

export default function UseLifecycleHooks () {
    useLifecycleHooks({
        onMount: () => console.log('lifecycle mounted'),
        onUnmount: () => console.log('lifecycle unmounting')

    })

    return <div>look at the console and click the button</div> }
```

Alternatively, we could use the two Hooks separately:

```
import React from 'react'
import { useOnMount, useOnUnmount } from 'react-hookedup'

export default function UseLifecycleHooksSeparate () {
    useOnMount(() => console.log('separate lifecycle mounted'))

    useOnUnmount(() => console.log('separate lifecycle unmounting'))

    return <div>look at the console and click the button</div> }
```

However, if you have this kind of pattern, I would recommend simply using the `useEffect` Hook, as follows:

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

export default function LifecycleHooksWithEffect () {
    useEffect(() => {
        console.log('lifecycle mounted with effect')
        return () => console.log('lifecycle unmounting with effect')
    }, [])

    return <div>look at the console and click the button</div> }
```

Using `useEffect`, we can put our whole effect into a single function, and then simply return a function for cleanup. This pattern is especially useful when we learn about making our own Hooks in the next labs.

Effects make us think differently about React components. We do not have to think about the life cycle of a component at all. Instead, we think about effects, dependencies, and the cleanup of effects.

# Step 4: The useMergeState Hook

The `useMergeState` Hook works similarly to the `useState` Hook. However, it does not replace the current state, but instead merges the current state with the new state, just like `this.setState()` works in React class components.

The Merge State Hook returns the following objects:

- `state`: The current state
- `setState`: A function to merge the current state with the given state object

For example, let's consider the following component:

1. First, we import the `useState` Hook:

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

2. Then, we define our app component and a State Hook with an object containing a `loaded` value and a `counter` value:

```
export default function MergeState () {
    const [ state, setState ] = useState({ loaded: true, counter: 0 })
```

3. Next, we define a `handleClick` function, where we set the new `state`, increasing the current `counter` value by `1`:

```
    function handleClick () {
        setState({ counter: state.counter + 1 })
    }
```

4. Finally, we render the current `counter` value and a **+1** button in order to increase the `counter` value by `1`. The button will be disabled if `state.loaded` is `false` or `undefined`:

```
    return (
        <div>
            Count: {state.counter}
            <button onClick={handleClick}
               disabled={!state.loaded}>+1</button>
        </div>
    )
}
```

As we can see, we have a simple counter app, showing the current count and a **+1** button. The **+1** button will only be enabled when the `loaded` value is set to `true`.

If we now click on the **+1** button, `counter` will increase from `0` to `1`, but the button will get disabled, because we have overwritten the current `state` object with a new `state` object.

To solve this problem, we would have to adjust the `handleClick` function as follows:

```
function handleClick () {
    setState({ ...state, counter: state.counter + 1 })
}
```

Alternatively, we could use the `useMergeState` Hook in order to avoid this problem altogether, and get the same behavior that we had with `this.setState()` in class components:

```
import React from 'react'
import { useMergeState } from 'react-hookedup'

export default function UseMergeState () {
    const { state, setState } = useMergeState({ loaded: true, counter: 0 })
```

As we can see, by using the `useMergeState` Hook, we can reproduce the same behavior that we had with `this.setState()` in class components. So, we do not need to use spread syntax anymore. However, often, it is better to simply use multiple State Hooks or a Reducer Hook instead.

# Exercise 4: Various useful Hooks(Chapter8_3)

In addition to life cycle Hooks, `react-hookedup` also provides Hooks for timers, checking the network status, and various other useful Hooks for dealing with, for example, arrays and input fields. We are now going to cover the rest of the Hooks that `react-hookedup` provides.

These Hooks are as follows:

- The `usePrevious` Hook, to get the previous value of a Hook or prop
- Timer Hooks, to implement intervals and timeouts
- The `useOnline` Hook, to check whether the client has an active internet connection
- Various data manipulation Hooks for dealing with booleans, arrays, and counters
- Hooks to deal with focus and hover events

## Step 1: The usePrevious Hook

The `usePrevious` Hook is a simple Hook that lets us get the previous value of a prop or Hook value. It will always store and return the previous value of any given variable, and it works as follows:

1. First, we import the `useState` and `usePrevious` Hooks:

```
import React, { useState } from 'react'
import { usePrevious } from 'react-hookedup'
```

2. Then, we define our `App` component, and a Hook in which we store the current `count` state:

```
export default function UsePrevious () {
    const [ count, setCount ] = useState(0)
```

3. Now, we define the `usePrevious` Hook, passing the `count` value from the State Hook to it:

```
    const prevCount = usePrevious(count)
```

The `usePrevious` Hook works with any variable, including component props and values from other Hooks.

4. Next, we define a handler function, which will increase `count` by `1`:

```
    function handleClick () {
        setCount(count + 1)
    }
```

5. Finally, we render the previous value of `count`, the current value of `count`, and a button to increase `count`:

```
    return (
        <div>
            Count was {prevCount} and is {count} now.
            <button onClick={handleClick}>+1</button>
        </div>
    )
}
```

The previously defined component will first show **Count was and is 0 now.**, because the default value for the Previous Hook is `null`. When clicking the button once, it will show the following: **Count was 0 and is 1 now.**.

# Timer Hooks

The `react-hookedup` library also provides Hooks for dealing with timers. If we simply create a timer using `setTimeout` or `setInterval` in our component, it will get instantiated again every time the component is re-rendered. This not only causes bugs and unpredictability, but can also cause a memory leak if the old timers are not freed properly. Using timer Hooks, we can avoid these problems completely, and easily use intervals and timeouts.

The following timer Hooks are provided by the library:

- The `useInterval` Hook, which is used to define `setInterval` timers (timers that trigger multiple times) in React components
- The `useTimeout` Hook, which is used to define `setTimeout` timers (timers that trigger only once after a certain amount of time)

## Step 2: The useInterval Hook

The `useInterval` Hook can be used just like `setInterval`. We are now going to implement a small counter that counts the number of seconds since mounting the component:

1. First, import the `useState` and `useInterval` Hooks:

```
import React, { useState } from 'react'
import { useInterval } from 'react-hookedup'
```

2. Then, we define our component and a State Hook:

```
export default function UseInterval () {
    const [ count, setCount ] = useState(0)
```

3. Next, we define the `useInterval` Hook, which is going to increase the `count` by `1` every `1000` ms, which is equal to `1` second:

```
        useInterval(() => setCount(count + 1), 1000)
```

4. Finally, we display the current `count` value:

```
        return <div>{count} seconds passed</div> }
```

Alternatively, we could use an Effect Hook in combination with `setInterval`, instead of the `useInterval` Hook, as follows:

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

   export default function IntervalWithEffect () {     const [
   count, setCount ] = useState(0)
       useEffect(() => {
           const interval = setInterval(() => setCount(count + 1), 1000)
           return () => clearInterval(interval)

       })

       return <div>{count} seconds passed</div> }
```

As we can see, the `useInterval` Hook makes our code much more concise and easily readable.

## Step 3: useTimeout Hook

The `useTimeout` Hook can be used just like `setTimeout`. We are now going to implement a component that triggers after `10` seconds have passed: 1. First, import the `useState` and `useTimeout` Hooks:

```
import React, { useState } from 'react'
import { useTimeout } from 'react-hookedup'
```

2. Then, we define our component and a State Hook:

```
export default function UseTimeout () {
    const [ ready, setReady ] = useState(false)
```

3. Next, we define the `useTimeout` Hook, which is going to set `ready` to `true`, after `10000` ms (`10` seconds):

```
useTimeout(() => setReady(true), 10000)
```

4. Finally, we display whether we are ready or not:

```
return <div>{ready ? 'ready' : 'waiting...'}</div> }
```

Alternatively, we could use an Effect Hook in combination with `setTimeout`, instead of the `useTimeout` Hook, as follows:

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

export default function TimeoutWithEffect () {
    const [ ready, setReady ] = useState(false)
    useEffect(() => {
        const timeout = setTimeout(() => setReady(true), 10000)
        return () => clearTimeout(timeout)
    })

    return <div>{ready ? 'ready' : 'waiting...'}</div> }
```

As we can see, the `useTimeout` Hook makes our code much more concise and easily readable.

## Step 4: The Online Status Hook

In some web apps, it makes sense to implement an offline mode; for example, if we want to be able to edit and save drafts for posts locally, and sync them to the server whenever we are online again. To be able to implement this use case, we can use the `useOnlineStatus` Hook.

The Online Status Hook returns an object with an `online` value, which contains `true` if the client is online; otherwise, it contains `false`. It works as follows:

```
import React from 'react'
import { useOnlineStatus } from 'react-hookedup'

export default function App () {
    const { online } = useOnlineStatus()
    return <div>You are {online ? 'online' : 'offline'}!</div> }
```

The previous component will display **You are online!**, when an internet connection is available, or **You are offline!**, otherwise.

We could then use a Previous Hook, in combination with an Effect Hook, in order to sync data to the server when we are online again:

```
import React, { useEffect } from 'react'
import { useOnlineStatus, usePrevious } from 'react-hookedup'

export default function App () {

    const { online } = useOnlineStatus()
    const prevOnline = usePrevious(online)
    useEffect(() => {
        if (prevOnline === false && online === true) {
            alert('syncing data')
        }
    }, [prevOnline, online])

    return <div>You are {online ? 'online' : 'offline'}!</div> }
```

Now, we have an Effect Hook that triggers whenever the value of `online` changes. It then checks whether the previous value of `online` was `false`, and the current one is `true`. If that is the case, it means we were offline, and are now online again, so we need to sync our updated data to the server. As a result, our app will show an alert displaying **syncing data** when we go offline and then online again.

# Data manipulation Hooks

The `react-hookedup` library provides various utility Hooks for dealing with data. These Hooks simplify dealing with common data structures and provide an abstraction over the State Hook.

The following data manipulation Hooks are provided:

- The `useBoolean` Hook: To deal with toggling boolean values
- The `useArray` Hook: To deal with handling arrays
- The `useCounter` Hook: To deal with counters

## Step 1: The useBoolean Hook

The `useBoolean` Hook is used to deal with toggling boolean values (`true`/`false`), and provides functions to set the value to `true`/`false`, and a `toggle` function to toggle the value.

The Hook returns an object with the following:

- `value`: The current value of the boolean
- `toggle`: A function to toggle the current value (sets `true` if currently `false`, and `false` if currently `true`)
- `setTrue`: Sets the current value to `true`
- `setFalse`: Sets the current value to `false`

The Boolean Hook works as follows:

1. First, we import the `useBoolean` Hook from `react-hookedup`:

```
import React from 'react'
import { useBoolean } from 'react-hookedup'
```

2. Then, we define our component and the Boolean Hook, which returns an object with the `toggle` function and `value`. We pass `false` as the default value:

```
export default function UseBoolean () {
    const { toggle, value } = useBoolean(false)
```

3. Finally, we render a button, which can be turned **on**/**off**:

```
    return (
        <div>
            <button onClick={toggle}>{value ? 'on' :'off'}</button>
        </div>
    )
}
```

The button will initially be rendered with the text **off**. When clicking the button, it will show the text **on**. When clicking again, it will be **off** again.

## Step 2: The useArray Hook

The `useArray` Hook is used to easily deal with arrays, without having to use the rest/spread syntax.

The Array Hook returns an object with the following:

- `value`: The current array
- `setValue`: Sets a new array as the value
- `add`: Adds a given element to the array
- `clear`: Removes all elements from the array
- `removeIndex`: Removes an element from the array by its index
- `removeById`: Removes an element from the array by its `id` (assuming that the elements in the array are objects with an `id` key)

It works as follows:

1. First, we import the `useArray` Hook from `react-hookedup`:

```
import React from 'react'
import { useArray } from 'react-hookedup'
```

2. Then, we define the component and the Array Hook, with the default value of `['one', 'two', 'three']`:

```
export default function UseArray () {
    const { value, add, clear, removeIndex } = useArray(['one',
          'two', 'three'])
```

3. Now, we display the current array as JSON:

```
    return (
        <div>
            <p>current array: {JSON.stringify(value)}</p>
```

4. Then, we display a button to `add` an element:

```
            <button onClick={() => add('test')}>add element</button>
```

5. Next, we display a button to remove the first element by index:

```
            <button onClick={() => removeIndex(0)}>remove first
element</button>
```

6. Finally, we add a button to `clear` all elements:

```
            <button onClick={() => clear()}>clear elements</button>
        </div>
    )
}
```

As we can see, using the `useArray` Hook makes dealing with arrays much simpler.

## Step 3: The useCounter Hook

The `useCounter` Hook can be used to define various kinds of counters. We can define a lower/upper limit, specify whether the counter should loop or not, and specify the step amount by which we increase/decrease the counter. Furthermore, the Counter Hook provides functions in order to increase/decrease the counter.

It accepts the following configuration options:

- `upperLimit`: Defines the upper limit (maximum value) of our counter
- `lowerLimit`: Defines the lower limit (minimum value) of our counter
- `loop`: Specifies whether the counter should loop (for example, when the maximum value is reached, we go back to the minimum value)
- `step`: Sets the default step amount for the increase and decrease functions It returns the following object:
- `value`: The current value of our counter.
- `setValue`: Sets the current value of our counter.
- `increase`: Increases the value by a given step amount. If no amount is specified, then the default step amount is used.
- `decrease`: Decreases the value by a given step amount. If no amount is specified, then the default step amount is used.

The Counter Hook can be used as follows:

1. First, we import the `useCounter` Hook from `react-hookedup`:

```
import React from 'react'
import { useCounter } from 'react-hookedup'
```

2. Then, we define our component and the Hook, specifying `0` as the default value. We also specify `upperLimit`, `lowerLimit`, and `loop`:

```
export default function UseCounter () {
    const { value, increase, decrease } = useCounter(0, { upperLimit: 3,
lowerLimit: 0, loop: true })
```

3. Finally, we render the current value and two buttons to `increase`/`decrease` the value:

```
    return (
        <div>
            <b>{value}</b>
            <button onClick={increase}>+</button>
            <button onClick={decrease}>-</button>
        </div>
    )
}
```

As we can see, the Counter Hook makes implementing counters much simpler.

# Exercise 5: Focus and Hover Hooks

Sometimes, we want to check whether the user has hovered over an element or focused on an `input` field. To do so, we can use the Focus and Hover Hooks that are provided by the `react-hookedup` library.

The library provides two Hooks for these features:

- The `useFocus` Hook: To handle focus events (for example, a selected `input` field)
- The `useHover` Hook: To deal with hover events (for example, when hovering the mouse pointer over an area)

## Step 1: The useFocus Hook

In order to know whether an element is currently focused, we can use the `useFocus` Hook as follows:

1. First, we import the `useFocus` Hook:

```
import React from 'react'
import { useFocus } from 'react-hookedup'
```

2. Then, we define our component and the Focus Hook, which returns the `focused` value and a `bind` function, to bind the Hook to an element:

```
export default function UseFocus () {
    const { focused, bind } = useFocus()
```

3. Finally, we render an `input` field, and bind the Focus Hook to it:

```
    return (
        <div>
            <input {...bind} value={focused ? 'focused' : 'not
focused'} />
</div>
    )
}
```

As we can see, the Focus Hook makes it much easier to handle focus events. There is no need to define our own handler functions anymore.

## Step 2: The useHover Hook

In order to know whether the user is currently hovering over an element, we can use the `useHover` Hook, as follows:

1. First, we import the `useHover` Hook:

```
import React from 'react'
import { useHover } from 'react-hookedup'
```

2. Then, we define our component and the Hover Hook, which returns the `hovered` value and a `bind` function, to bind the Hook to an element:

```
export default function UseHover () {
    const { hovered, bind } = useHover()
```

3. Finally, we render an element, and bind the Hover Hook to it:

```
    return (
        <div {...bind}>Hover me {hovered && 'THANKS!!!'}</div>
    )
}
```

As we can see, the Hover Hook makes it much easier to handle hover events. There is no need to define our own handler functions anymore.

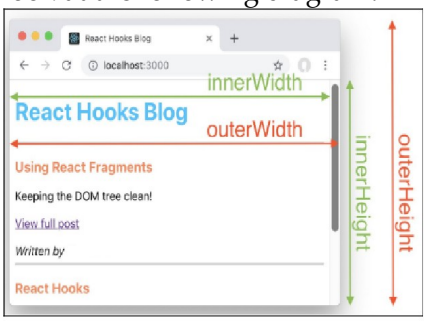
# Exercise 6: Responsive design with Hooks (Chapter8_4)

In web apps, it is often important to have a responsive design. Responsive design makes your web app render well on various devices and window/screen sizes. Our blog app might be viewed on a desktop, a mobile phone, a tablet, or maybe even a very large screen, such as a TV.
Often, it makes the most sense to simply use CSS media queries for responsive design. However, sometimes that is not possible, for example, when we render elements within a canvas or **Web Graphics Library** (**WebGL**). Sometimes, we also want to use the window size in order to decide whether to load a component or not, instead of simply rendering it and then hiding it via CSS later.

The `@rehooks/window-size` library provides the `useWindowSize` Hook, which returns the following values:

- `innerWidth`: Equal to the `window.innerWidth` value
- `innerHeight`: Equal to the `window.innerHeight` value
- `outerWidth`: Equal to the `window.outerWidth` value
- `outerHeight`: Equal to the `window.outerHeight` value

To show the difference between `outerWidth`/`outerHeight`, and `innerWidth`/`innerHeight`, take a look at the following diagram:




Visualization of the window width/height properties

As we can see, `innerHeight` and `innerWidth` specify the innermost part of the browser window, while `outerHeight` and `outerWidth` specify the full dimensions of the browser window, including the URL bar, scroll bars, and so on.

We are now going to hide components based on the window size in our blog app.

# Responsively hiding components

In our blog app, we are going to hide the `UserBar` and `ChangeTheme` components completely when the screen size is very small so that, when reading a post on a mobile phone, we can focus on the content.

**Step 1:** Let's get started implementing the Window Size Hook:

1. First, we have to install the `@rehooks/window-size` library:

```
> npm install --save @rehooks/window-size
```

2. Then, we import the `useWindowSize` Hook at the start of the **`src/pages/HeaderBar.js`** file:

```
import useWindowSize from '@rehooks/window-size'
```

3. Next, we define the following Window Size Hook after the existing Context Hooks:

```
const { innerWidth } = useWindowSize()
```

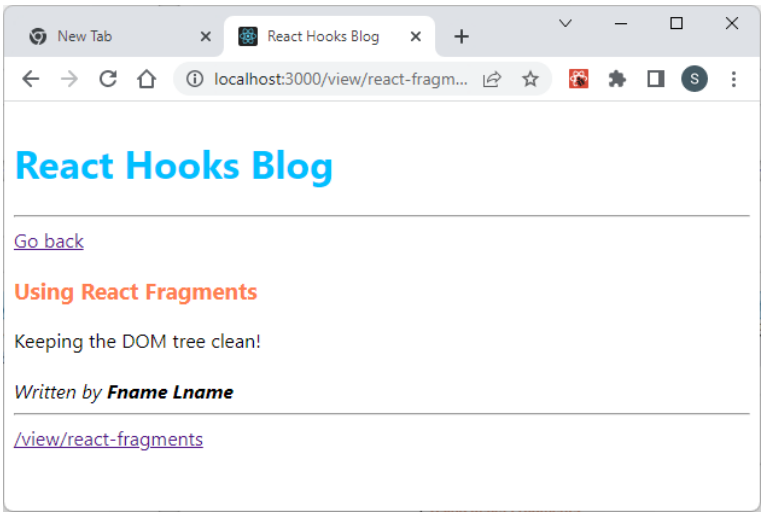
4. If the window width is smaller than `640` pixels, we assume that the device is a mobile phone:

```
const mobilePhone = innerWidth < 640
```

5. Finally, we only show the `ChangeTheme` and `UserBar` components when we are not on a mobile phone:

```
              {!mobilePhone && <ChangeTheme theme={theme}
setTheme={setTheme} />}
              {!mobilePhone && <br />}
              {!mobilePhone && <React.Suspense
fallback={"Loading..."}>
                  <UserBar />
              </React.Suspense>}
              {!mobilePhone && <br />}
```

If we now resize our browser window to a width smaller than `640` pixels, we can see that the `ChangeTheme` and `UserBar` components will not be rendered anymore:




Using the Window Size Hook, we can avoid rendering elements on smaller screen sizes.

# Exercise 7: Undo/Redo with Hooks(Chapter8_5)

In some apps, we want to implement undo/redo functionality, which means that we can go back and forth in the state of our app. For example, if we have a text editor in our blog app, we want to provide a feature to undo/redo changes. If you learned about Redux, you might already be familiar with this kind of functionality. Since React now provides a Reducer Hook, we can reimplement the same functionality using only React. The `use-undo` library provides exactly this functionality.
The `useUndo` Hook takes the default `state` object as an argument, and returns an array with the following contents: `[ state, functions ]`.

The `state` object looks as follows:

- `present`: The current state
- `past`: Array of past states (when we undo, we go here)
- `future`: Array of future states (after undoing, we can redo to go here)

The `functions` object returns various functions to interact with the Undo Hook:

- `set`: Sets the current state, and assigns a new value to `present`.
- `reset`: Resets the current state, clears the `past` and `future` arrays (undo/redo history), and assigns a new value to `present`.
- `undo`: Undoes to the previous state (goes through the elements of the `past` array). `redo`: Redoes to the next state (goes through the elements of the `future` array).
- `canUndo`: Equals `true` if it is possible to do an undo action (`past` array not empty).
- `canRedo`: Equals `true` if it is possible to do a redo action (`future` array not empty).

We are now going to implement undo/redo functionality in our post editor.

# Step 1: Implementing Undo/Redo in our post editor

In the simple post editor of our blog app, we have a `textarea` where we can write the contents of a blog post. We are now going to implement the `useUndo` Hook there, so that we can undo/redo any changes that we made to the text:

1. First, we have to install the `use-undo` library via `npm`:

```
> npm install --save use-undo
```

2. Then, we import the `useUndo` Hook from the library in **`src/post/CreatePost.js`**:

```
import useUndo from 'use-undo'
```

3. Next, we define the Undo Hook by replacing the current `useInput` Hook. Remove the following line of code:

```
const { value: content, bindToInput: bindContent } = useInput('')
```

Replace it with the `useUndo` Hook, as follows. We set the default state to `''`. We also save the state to `undoContent`, and get the `setContent`, `undo`, and `redo` functions, as well as the `canUndo` and `canRedo` values:

```
const [ undoContent, {
    set: setContent,
    undo,
    redo,
    canUndo,
    canRedo
} ] = useUndo('')
```

4. Now, we assign the `undoContent.present` state to the `content` variable:

```
const content = undoContent.present
```

5. Next, we define a new handler function in order to update the `content` value using the `setContent` function:

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

6. Then, we have to replace the `bindContent` object with the `handleContent` function, as follows:

```
<textarea value={content} onChange={handleContent} />
```

7. Finally, we define buttons to **Undo**/**Redo** our changes, after the `textarea` element:

```
            <button type="button" onClick={undo}
disabled={!canUndo}>Undo</button>
            <button type="button" onClick={redo}
disabled={!canRedo}>Redo</button>
```

It is important that `<button>` elements in a `<form>` element have a `type` attribute defined. If the `type` attribute is not defined, buttons are assumed to be `type="submit"`, which means that they will trigger the `onSubmit` handler function when clicked.

Now, after entering text we can press **Undo** to remove one character at a time, and **Redo** to add the characters again. Next, we are going to implement debouncing, which means that our changes will only be added to the undo history after a certain amount of time, not after every character that we entered.

# Debouncing with Hooks

As we have seen in the previous section, when we press **Undo**, it undoes a single character at a time. Sometimes, we do not want to store every change in our undo history. To avoid storing every change, we need to implement debouncing, which means that the function that stores our `content` to the undo history is only called after a certain amount of time.

The `use-debounce` library provides the `useDebounce` Hook, which can be used, as follows, for simple values:

```
const [ text, setText ] = useState('')

const [ value ] = useDebounce(text, 1000)
```

Now, if we change the text via `setText`, the `text` value will be updated instantly, but the `value` variable will only be updated after `1000` ms (`1` second).

However, for our use case, this is not enough. We are going to need debounced callbacks in order to implement debouncing in combination with `use-undo`. The `usedebounce` library also provides the `useDebouncedCallback` Hook, which can be used as follows:

```
const [ text, setText ] = useState('')
const [ debouncedSet, cancelDebounce ] = useDebouncedCallback(
    (value) => setText(value), 1000
)
```

Now, if we call `debouncedSet('text')`, the `text` value will be updated after `1000` ms (`1` second). If `debouncedSet` is called multiple times, the timeout will get reset every time, so that only after `1000` ms of no further calls to the `debouncedSet` function, the `setText` function will be called. Next, we are going to move on to implementing debouncing in our post editor.

# Step 2: Debouncing changes in our post editor

Now that we have learned about debouncing, we are going to implement it in combination with the Undo Hook in our post editor, as follows:

1. First, we have to install the `use-debounce` library via `npm`:

```
> npm install --save use-debounce
```

2. In **`src/post/CreatePost.js`**, first make sure that you import the `useState` Hook, if it is not imported already:

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

3. Next, import the `useDebouncedCallback` Hook from the `use-debounce` library:

```
import { useDebouncedCallback } from 'use-debounce'
```

4. Now, before the Undo Hook, define a new State Hook, which we are going to use for the non-debounced value, to update the `input` field: \

```
const [ content, setInput ] = useState('')
```

5. After the Undo Hook, we remove the assignment of the `content` value. Remove the following code:

```
const content = undoContent.present
```

6. Now, after the Undo Hook, define the Debounced Callback Hook:

```
const [ setDebounce, cancelDebounce ] = useDebouncedCallback(
```

7. Within the Debounced Callback Hook, we define a function in order to set the content of the Undo Hook:

```
(value) => {
    setContent(value)
},
```

8. We trigger the `setContent` function after `200` ms:

```
    200
)
```

9. Next, we have to define an Effect Hook, which will trigger whenever the undo state changes. In this Effect Hook, we cancel the current debouncing, and set the `content` value to the current `present` value:

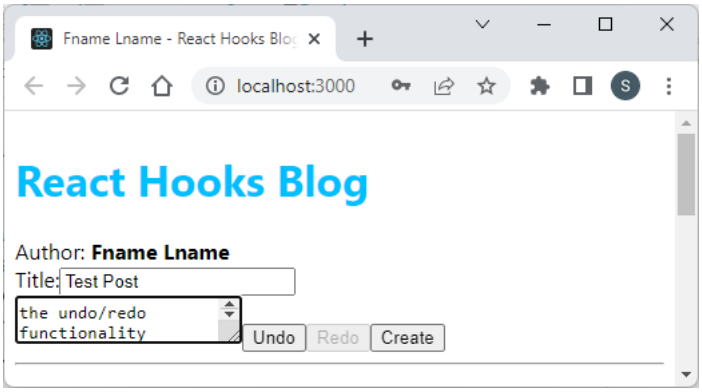
```
useEffect(() => {
    cancelDebounce()
    setInput(undoContent.present)

}, [undoContent])
```

10. Finally, we adjust the `handleContent` function in order to trigger the `setInput` function, as well as the `setDebounce` function:

```
function handleContent (e)
    const { value } = e.target
    setInput(value)
    setDebounce(value)
}
```

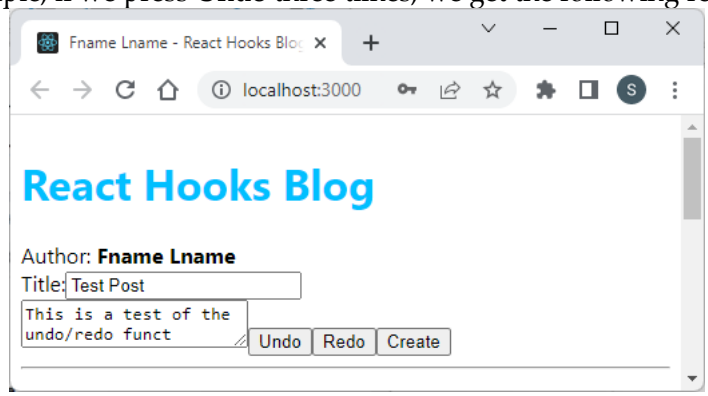
As a result, we instantly set the input `value`, but we do not store anything to the undo history yet. After the debouncing callback triggers (after `200` ms), we store the current value to the undo history. Whenever the undo state updates, for example, when we press the **Undo**/**Redo** buttons, we cancel the current debouncing to avoid overwriting the value after undoing/redoing. Then, we set the `content` value to the new `present` value of the Undo Hook.

If we now type some text into our editor, we can see that the **Undo** button only activates after a while. It then looks like this:




Undo button activated after typing some text

If we now press the **Undo** button, we can see that we will not undo character by character, but more text at once. For example, if we press **Undo** three times, we get the following result:




As we can see, **Undo**/**Redo** and debouncing now work perfectly fine!