10 Building Your Own Hooks

Exercise 1: Creating a useTheme Hook

In many components, we use the `ThemeContext` to style our blog app. Functionality that is used across multiple components is usually a good opportunity for creating a custom Hook. As you might have noticed, we often do the following:

```
import { ThemeContext } from '../contexts'
export default function SomeComponent () {
    const theme = useContext(ThemeContext)
    // ...
```

We could abstract this functionality into a `useTheme` Hook, which will get the `theme` object from the `ThemeContext`.

Step 1: Let's start creating a custom `useTheme` Hook:

1. Create a new `src/hooks/` directory, which is where we are going to put our custom Hooks.
2. Create a new `src/hooks/useTheme.js` file.
3. In this newly created file, we first import the `useContext` Hook and the `ThemeContext` as follows:

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

4. Next, we export a new function called `useTheme`; this will be our custom Hook. Remember, Hooks are just functions prefixed with the `use` keyword: `export`

   ```
   default function useTheme () {
   ```

5. In our custom Hook, we can now use the essential Hooks provided by React to build our own Hook. In our case, we simply return the `useContext` Hook:

   ```
       return useContext(ThemeContext) }
   ```

As we can see, custom Hooks can be quite simple. In this case, the custom Hook only returns a Context Hook with the `ThemeContext` passed to it. Nevertheless, this makes our code more concise and easier to change later. Furthermore, by using a `useTheme` Hook, it is clear that we want to access the theme, which means our code will be easier to read and reason about.

Exercise 2: Creating global state Hooks

Another thing that we often do is access the global state. For example, some components need the `user` state and some need the `posts` state. To abstract this functionality, which will also make it easier to adjust the state structure later on, we can create custom Hooks to get certain parts of the state:

- `useUserState`: Gets the `user` part of the `state` object
- `usePostsState`: Gets the `posts` part of the `state` object

Defining the useUserState Hook

Repeating a similar process to what we did for the `useTheme` Hook, we import the `useContext` Hook from React and the `StateContext`. However, instead of returning the result of the Context Hook, we now pull out the `state` object via destructuring and then return `state.user`.

Step 1: Create a new `src/hooks/useUserState.js` file with the following contents:

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

export default function useUserState () {
    const { state } = useContext(StateContext)
    return state.user }
```

Similarly to the `useTheme` Hook, the `useUserState` Hook makes our code more concise, easier to change later, and improves readability.

Step 2: Defining the usePostsState Hook

We repeat the same process for the `posts` state. Create a new `src/hooks/usePostsState.js` file with the following contents:

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

export default function usePostsState () {
    const { state } = useContext(StateContext)
    return state.posts
}
```

Similarly to the `useTheme` and `useUserState` Hooks, the `usePostsState` Hook makes our code more concise, easier to change later, and improves readability.

Exercise 3: Creating a useDispatch Hook

In many components, we need the `dispatch` function to do certain actions, so we often have to do the following:

```
import { StateContext } from '../contexts'
export default function SomeComponent () {
    const { dispatch } = useContext(StateContext)      // ...
```

We can abstract this functionality into a `useDispatch` Hook, which will get the `dispatch` function from our global state context. Doing this will also make it easier to replace the state management implementation later on. For example, later on, we could replace our simple Reducer Hook with a state management library such as Redux or MobX.

Step 1: Let's define the `useDispatch` Hook now using the following steps:

1. Create a new `src/hooks/useDispatch.js` file.
2. Import the `useContext` Hook from React and the `StateContext` as follows:

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

3. Next, we define and export the `useDispatch` function; here, we allow passing a different `context` as an argument for making the Hook more generic (in case we want to use the `dispatch` function from a local state context later on). However, we set the default value of the `context` argument to the `StateContext` like so:

   ```
   export default function useDispatch (context = StateContext) {
   ```

4. Finally, we pull out the `dispatch` function from the Context Hook via destructuring and return it with the following code:

   ```
       const { dispatch } = useContext(context)
       return dispatch
   }
   ```

As we can see, creating a custom Dispatch Hook makes our code easier to change later on, as we only need to adjust the `dispatch` function in one place.

Exercise 4: Creating API Hooks

We can also create Hooks for the various API calls. Putting these Hooks in a single file allows us to adjust the API calls easily later on. We are going to prefix our custom API Hooks with `useAPI` so it is easy to tell which functions are API Hooks.

Step 1: Let's create custom Hooks for our API now using the following steps:

1. Create a new `src/hooks/api.js` file.
2. Import the `useResource` Hook from the `react-request-hook` library as follows:
   ```
   import { useResource } from 'react-request-hook'
   ```
3. First, we define a `useAPILogin` Hook to log in a user; we simply cut and paste the existing code from the `src/user/Login.js` file like so:
   ```
   export function useAPILogin () {
       return useResource((username, password) => ({
           url:
   `/login/${encodeURI(username)}/${encodeURI(password)}`,
           method: 'get'
       }))
   }
   ```
4. Next, we define a `useAPIRegister` Hook; we simply cut and paste the existing code from the `src/user/Register.js` file as follows:
   ```
   export function useAPIRegister () {
       return useResource((username, password) => ({
           url: '/users',
           method: 'post',
           data: { username, password }
       }))
   }
   ```
5. Now we define a `useAPICreatePost` Hook, cutting and pasting the existing code from the `src/post/CreatePost.js` file, as follows:
   ```
   export function useAPICreatePost () {
       return useResource(({ title, content, author }) => ({
           url: '/posts',
   method: 'post',
           data: { title, content, author }
       }))
   }
   ```
6. Finally, we define a `useAPIThemes` Hook, cutting and pasting the existing code from the `src/ChangeTheme.js` file as follows:
   ```
   export function useAPIThemes () {
       return useResource(() => ({
           url: '/themes',
           method: 'get'
       }))
   }
   ```

As we can see, having all API-related functionality in one place makes it easier to adjust our API code later on.

Exercise 5: Creating a useDebouncedUndo Hook

We are now going to create a slightly more advanced Hook for debounced undo functionality. We already implemented this functionality in the `CreatePost` component. Now, we are going to extract this functionality into a custom `useDebouncedUndo` Hook.

Step 1: Let's create the `useDebouncedUndo` Hook with the following steps:

1. Create a new `src/hooks/useDebouncedUndo.js` file.
2. Import the `useState`, `useEffect`, and `useCallback` Hooks from React, as well as the `useUndo` Hook and the `useDebouncedCallback` Hook:

```
import { useState, useEffect, useCallback } from 'react'
import useUndo from 'use-undo'
import { useDebouncedCallback } from 'use-debounce'
```

3. Now we are going to define the `useDebouncedUndo` function, which accepts a `timeout` argument for the debounced callback: `export default function useDebouncedUndo (timeout = 200) {`

4. In this function, we copy over the `useState` Hook from the previous implementation, as shown here:

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

5. Next, we copy over the `useUndo` Hook; however, this time, we store all other undo-related functions in an `undoRest` object:

```
const [ undoContent, { set: setContent, ...undoRest } ] = useUndo('')
```

6. Then we copy over the `useDebouncedCallback` Hook, replacing the fixed `200` value with our `timeout` argument:

```
const [ setDebounce, cancelDebounce ] = useDebouncedCallback(
    (value) => {
        setContent(value)
    },
    timeout

)
```

7. Now we copy over the Effect Hook, as shown in the following code:

```
useEffect(() => {
    cancelDebounce()
    setInput(undoContent.present)
}, [cancelDebounce, undoContent])
```

8. Then, we define a `setter` function, which is going to set a new input `value` and call `setDebounce`. We can wrap the `setter` function with a `useCallback` Hook here to return a memoized version of the function and avoid recreating the function every time the component that uses the Hook re-renders. Similar to the `useEffect` and `useMemo`

Hooks, we also pass a dependency array as the second argument of the `useCallback` Hook:

```
    const setter = useCallback(function setterFn (value) {
        setInput(value)
        setDebounce(value)
    }, [ setInput, setDebounce ])
```

9. Finally, we return the `content` variable (containing the current input `value`), the `setter` function, and the `undoRest` object (which contains the `undo`/`redo` functions and the `canUndo`/`canRedo` booleans):

```
    return [ content, setter, undoRest ]
}
```

Creating a custom Hook for debounced undo means that we can reuse that functionality across multiple components. We could even provide this Hook as a public library, allowing others to easily implement debounced undo/redo functionality.

Exercise 6: Exporting our custom Hooks

After creating all our custom Hooks, we are going to create an `index.js` file in our Hooks directory and re-export our Hooks there, so that we can import our custom Hooks as follows:

```
import { useTheme } from './hooks'
```

Step 1: Let's export all our custom Hooks now using the following steps:

1. Create a new `src/hooks/index.js` file.
2. In this file, we first import our custom Hooks as follows:

   ```
   import useTheme from './useTheme'
   import useDispatch from './useDispatch'
   import usePostsState from './usePostsState'
   import useUserState from './useUserState'
   import useDebouncedUndo from './useDebouncedUndo'
   ```

3. Then, we re-export these imported Hooks with the following code:

   ```
   export { useTheme, useDispatch, usePostsState, useUserState,
   useDebouncedUndo }
   ```

4. Finally, we re-export all Hooks from the `api.js` file as follows:

   ```
   export * from './api'
   ```

Now that we have exported all our custom Hooks, we can simply import Hooks directly from the `hooks` folder, making it easier to import multiple custom Hooks at once.

Exercise 7: Using our custom Hooks (Chapter10_2)

After creating our custom Hooks, we can now start using them throughout our blog application. Using custom Hooks is quite straightforward as they are similar to community Hooks. Just like all other Hooks, custom Hooks are simply JavaScript functions.

We created the following Hooks:

- `useTheme`
- `useDispatch`
- `usePostsState`
- `useUserState`
- `useDebouncedUndo`
- `useAPILogin`
- `useAPIRegister`
- `useAPICreatePost`
- `useAPIThemes`

In this section, we are going to refactor our app to use all of our custom Hooks.

Using the useTheme Hook

Instead of using the `useContext` Hook with the `ThemeContext`, we can now use the `useTheme` Hook directly! If we end up changing the theming system later on, we can simply modify the `useTheme` Hook and our new system will be implemented throughout our application.

Step 1: Let's refactor our app to use the `useTheme` Hook:

1. Edit `src/Header.js` and replace the existing imports with an import of the `useTheme` Hook. The `ThemeContext` and `useContext` imports can be removed:

   ```
   import { useTheme } from './hooks'
   ```

2. Then, replace the current Context Hook definition with the `useTheme` Hook, as shown here:

   ```
       const { primaryColor } = useTheme()
   ```

3. Now edit `src/post/Post.js` and adjust the imports similarly there:

   ```
   import { useTheme } from './hooks'
   ```

4. Then, replace the `useContext` Hook with the `useTheme` Hook as follows:

   ```
   const { secondaryColor } = useTheme()
   ```

As we can see, using a custom Hook makes our code much more concise and easier to read. We now move on to using the global state Hooks.

Using the global state Hooks

Similarly to what we did with the `ThemeContext`, we can also replace our state Context Hooks with the `usePostsState`, `useUserState`, and `useDispatch` Hooks. This is optimal if we want to change the state logic later. For example, if our state grows and we want to use a more sophisticated system such as Redux or MobX, then we can simply adjust the existing Hooks and everything will work the same way as before.

In this section, we are going to adjust the following components:

- `UserBar`
- `Login`
- `Register`
- `Logout`
- `CreatePost`
- `PostList`

Step 2: Adjusting the UserBar component

First, we are going to adjust the `UserBar` component. Here, we can use the `useUserState` Hook by following these steps:

1. Edit `src/user/UserBar.js` and import the `useUserState` Hook:

```
import { useUserState } from '../hooks'
```

2. Then, we remove the following Hook definition:

```
    const { state } = useContext(StateContext)
    const { user } = state
```

3. We replace it with our custom `useUserState` Hook:

```
    const user = useUserState()
```

Now the `UserBar` component makes use of our custom Hook instead of directly accessing the `user` state.

Step 3: Adjusting the Login component

Next, we are going to adjust the `Login` component, where we can use the `useDispatch` Hook. This process is outlined in the following steps:

1. Edit `src/user/Login.js` and import the `useDispatch` Hook, as follows:

```
import { useDispatch } from '../hooks'
```

2. Then remove the following Context Hook:

```
const { dispatch } = useContext(StateContext)
```

3. Replace it with our custom `useDispatch` Hook:

```
const dispatch = useDispatch()
```

Now the `Login` component makes use of our custom Hook instead of directly accessing the `dispatch` function. Next, we are going to adjust the `Register` component.

Step 4: Adjusting the Register component

Similarly to the `Login` component, we can also use the `useDispatch` Hook in the `Register` component, as shown in the following steps:

1. Edit `src/user/Register.js` and import the `useDispatch` Hook:

```
import { useDispatch } from '../hooks'
```

2. Then, replace the current Context Hook with our custom Dispatch Hook, as shown here:

```
const dispatch = useDispatch()
```

Now the `Register` component also makes use of our custom Hook instead of directly accessing the `dispatch` function.

Step 5: Adjusting the Logout component

Then, we are going to adjust the `Logout` component to use both the `useUserState` and the `useDispatch` Hooks with the following steps:

1. Edit `src/user/Logout.js` and import the `useUserState` and `useDispatch` Hooks:

```
import { useDispatch, useUserState } from '../hooks'
```

2. Then, replace the current Hook definitions with the following:

```
const dispatch = useDispatch()
const user = useUserState()
```

Now the `Logout` component makes use of our custom Hooks instead of directly accessing the `user` state and the `dispatch` function.

Step 6: Adjusting the CreatePost component

Next we are going to adjust the `CreatePost` component, which is similar to what we did with the `Logout` component. This process is outlined in the following steps:

1. Edit `src/post/CreatePost.js` and import the `useUserState` and `useDispatch` Hooks:

   ```
   import { useUserState, useDispatch } from '../hooks'
   ```

2. Then, replace the current Context Hook definition with the following:

   ```
   const user = useUserState()

   const dispatch = useDispatch()
   ```

Now the `CreatePost` component makes use of our custom Hooks instead of directly accessing the `user` state and the `dispatch` function.

Step 7: Adjusting the PostList component

Finally, we are going to use the `usePostsState` Hook to render the `PostList` component, as follows:

1. Edit `src/post/PostList.js` and import the `usePostsState` Hook:

   ```
   import { usePostsState } from '../hooks'
   ```

2. Then replace the current Hook definition with the following:

   ```
   const posts = usePostsState()
   ```

Now the `PostList` component makes use of our custom Hook instead of directly accessing the `posts` state.

Exercise 8: Using the API Hooks

Next, we are going to replace all the `useResource` Hooks with our custom API Hooks. Doing so allows us to have all the API calls in one file so that we can easily adjust them later on, in case the API changes.

In this section, we are going to adjust the following components:

- `ChangeTheme`
- `Register`
- `Login`
- `CreatePost`

Let's get started.

Step 1: Adjusting the ChangeTheme component

First, we are going to adjust the `ChangeTheme` component and replace the Resource Hook, accessing `/themes` with our custom `useAPIThemes` Hook in the following steps:

1. In `src/ChangeTheme.js`, remove the following `useResource` Hook import statement:

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

Replace it with our custom `useAPIThemes` Hook: `import`

```
{ useAPIThemes } from './hooks'
```

2. Then, replace the `useResource` Hook definition with the following custom Hook:

```
const [ themes, getThemes ] = useAPIThemes()
```

Now the `ChangeTheme` component uses our custom API Hook to pull themes from the API.

Step 2: Adjusting the Register component

Next, we are going to adjust the `Register` component with the following steps:

1. Edit `src/user/Register.js` and adjust the import statement to also import the `useAPIRegister` Hook:

```
import { useDispatch, useAPIRegister } from '../hooks'
```

2. Then, replace the current Resource Hook with the following:

```
const [ user, register ] = useAPIRegister()
```

Now the `Register` component uses our custom API Hook to `register` users via the API.

Step 3: Adjusting the Login component

Similar to the `Register` component, we are also going to adjust the `Login` component:

1. Edit `src/user/Login.js` and adjust the import statement to also import the `useAPILogin` Hook:

```
import { useDispatch, useAPILogin } from '../hooks'
```

2. Then, replace the current Resource Hook with the following:

```
const [ user, login ] = useAPILogin()
```

Now the `Login` component uses our custom API Hook to log in users via the API.

Step 4: Adjusting the CreatePost component

Finally, we are going to adjust the `CreatePost` component by following these steps:

1. Edit `src/post/CreatePost.js` and adjust the import statement to also import the `useAPICreatePost` Hook:

```
import { useUserState, useDispatch, useAPICreatePost } from '../hooks'
```

2. Then, replace the current Resource Hook with the following:

```
const [ post, createPost ] = useAPICreatePost()
```

Now the `CreatePost` component uses our custom API Hook to create new posts via the API.

Step 5: Using the useDebouncedUndo Hook

Finally, we are going to replace all debounced undo logic in the `src/post/CreatePost.js` file with our custom `useDebouncedUndo` Hook. Doing so will make our component code much cleaner and easier to read. Furthermore, we can reuse the same debounced undo functionality in other components later.

Let's get started using the Debounced Undo Hook in the `CreatePost` component by following these steps:

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

```
import { useUserState, useDispatch, useDebouncedUndo, useAPICreatePost }
from '../hooks'
```

2. Then, remove the following code related to debounced undo handling:

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

const [ setDebounce, cancelDebounce ] = useDebouncedCallback(
    (value) => {
        setContent(value)
    },
    200
)
useEffect(() =>
{        cancelDebounce()

    setInput(undoContent.present)

}, [cancelDebounce, undoContent])
```

Replace it with our custom `useDebouncedUndo` Hook, as follows:

```
const [ content, setContent, { undo, redo, canUndo, canRedo } ]
= useDebouncedUndo()
```

3. Finally, remove the following setter functions in our `handleContent` function (marked in bold):

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

We can now use the `setContent` function provided by our custom Hook instead:

```
function handleContent (e) {

    const { value } = e.target

    setContent(value)

}
```

As you can see, our code is much cleaner, more concise, and easier to read now. Furthermore, we can reuse the Debounced Undo Hook in other components later on.

Interactions between Hooks

Our whole blog app now works in the same way as before, but it uses our custom Hooks! Until now, we have always had Hooks that encapsulated the whole logic, with only constant values being passed as arguments to our custom Hooks. However, we can also pass values of other Hooks into custom Hooks!

> Since Hooks are simply JavaScript functions, all Hooks can accept any value as arguments and work with them: constant values, component props, or even values from other Hooks.

We are now going to create local Hooks, which means that they will be placed in the same file as the component, because they are not needed anywhere else. However, they will still make our code easier to read and maintain. These local Hooks will accept values from other Hooks as arguments.

The following local Hooks will be created:

- A local Register Effect Hook
- A local Login Effect Hook

Let's see how to create them in the following subsections.

Exercise 9: Creating a local Register Effect Hook (Chapter10_3)

First of all, we are going to extract the Effect Hook from our `Login` component to a separate `useRegisterEffect` Hook function. This function will accept the following values from other Hooks as arguments: `user` and `dispatch`.

Step 1: Let's create a local Effect Hook for the `Register` component now using the following steps:

1. Edit `src/user/Register.js` and define a new function outside of the component function, right after the import statements:

```
function useRegisterEffect (user, dispatch) {
```

2. For the contents of the function, cut the existing Effect Hook from the `Register` component and paste it here:

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

3. Finally, define our custom `useLoginEffect` Hook where we cut out the previous Effect Hook, and pass the values from the other Hooks to it:

```
useRegisterEffect(user, dispatch)
```

As we can see, extracting an effect into a separate function makes our code easier to read and maintain.

Exercise 10: Creating a local Login Effect Hook

Repeating a similar process to the local Register Effect Hook, we are also going to extract the Effect Hook from our `Login` component to a separate `useLoginEffect` Hook function. This function will accept the following values from other Hooks as arguments: `user`, `dispatch`, and `setLoginFailed`.

Step 1: Let's create a local Hook for the `Login` component now using the following steps:

1. Edit `src/user/Login.js` and define a new function outside of the component function, right after the import statements:

```
function useLoginEffect (user, dispatch, setLoginFailed) {
```

2. For the contents of the function, cut the existing Effect Hook from the `Login` component and paste it here:

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

Here, we also added setLoginFailed to the Effect Hook dependencies. This is to make sure that whenever the setter function changes (which could happen eventually when using the Hook) the Hook triggers again. Always passing all dependencies of an Effect Hook, including functions, prevents bugs and unexpected behavior later on.

3. Finally, define our custom `useLoginEffect` Hook, where we cut out the previous Effect Hook, and pass the values from the other Hooks to it:

```
useLoginEffect(user, dispatch, setLoginFailed)
```

As we can see, extracting an effect into a separate function makes our code easier to read and maintain.

Testing Hooks

Exercise 11: Using the React Hooks Testing Library (Chapter10_4)

Step 1: In addition to the React Hooks Testing Library, we also need a special renderer for React. To render React components to the DOM, we used `react-dom`; for tests, we can use the `react-test-renderer`. We are now going to install the React Hooks Testing Library and the `react-test-renderer` via `npm`:

```
> npm install --save-dev @testing-library/react-hooks react-test-renderer
```

The React Hooks Testing Library should be used in the following circumstances:

- When writing libraries that define Hooks
- When you have Hooks that are used throughout multiple components (global Hooks)

However, the library should not be used when a Hook is only defined and used in a single component (local Hooks).

In that case, we should test the component directly using the React Testing Library. However, testing React components is beyond the scope of this book. More information about testing components can be found on the library website:
`https://testing-library.com/docs/react  -testing-library/intro`.

Step 2: Creating the useCounter Hook

The `useCounter` Hook is going to provide a current `count` and functions to `increment` and `reset` the counter.

Let's create the `useCounter` Hook now using the following steps:

1. Create a new `src/hooks/useCounter.js` file.
2. Import the `useState` and `useCallback` Hooks from React as follows:

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

3. We define a new `useCounter` Hook function with an argument for the `initialCount`: `export default function useCounter (initialCount = 0) {`

4. Then, we define a new State Hook for the `count` value with the following code:

```
    const [ count, setCount ] = useState(initialCount)
```

5. Next, we define functions for incrementing and resetting the `count`, as shown here:

```
    const increment = useCallback(() => setCount(count + 1), [])

    const reset = useCallback(() => setCount(initialCount),
[initialCount])
```

6. Finally, we return the current `count` and the two functions:

```
    return { count, increment, reset } }
```

Now that we have defined a simple Hook, we can start testing it.

Step 3: Testing the useCounter Hook result

Let's now write tests for the `useCounter` Hook we created, by following these steps:

1. Create a new `src/hooks/useCounter.test.js` file.
2. Import the `renderHook` and `act` functions from the React Hooks Testing Library, as we are going to use these later:

   ```
   import { renderHook, act } from '@testing-library/react-hooks'
   ```

3. Also, import the to-be-tested `useCounter` Hook, as shown here:

   ```
   import useCounter from './useCounter'
   ```

4. Now we can write our first test. To define a test, we use the `test` function from Jest. The first argument is the name of the test and the second argument is a function to be run as the test:

   ```
   test('should use counter', () => {
   ```

5. In this test, we use the `renderHook` function to define our Hook. This function returns an object with a `result` key, which is going to contain the result of our Hook:

   ```
       const { result } = renderHook(() => useCounter())
   ```

6. Now we can check the values of the `result` object using `expect` from Jest. The `result` object contains a `current` key, which will contain the current result from the Hook:

   ```
       expect(result.current.count).toBe(0)
       expect(typeof result.current.increment).toBe('function') })
   ```

As we can see, writing tests for Hook results is quite simple! When creating custom Hooks, especially when they are going to be used publicly, we should always write tests to ensure they work correctly.

Step 4: Testing useCounter Hook actions

Using the `act` function from the React Hooks Testing Library, we can execute functions from the Hook and then check the new result.

Let's now test actions of our Counter Hook:

1. Write a new `test` function, as shown in the following code:

   ```
   test('should increment counter', () => {
       const { result } = renderHook(() => useCounter())
   ```

2. Call the `increment` function of the Hook within the `act` function:

   ```
       act(() => result.current.increment())
   ```

3. Finally, we check whether the new `count` is now `1`:

   ```
       expect(result.current.count).toBe(1) })
   ```

As we can see, we can simply use the `act` function to trigger actions in our Hook and then test the value just like we did before.

Step 5: Testing the useCounter initial value

We can also check the result before and after calling `act` and pass an initial value to our Hook.

Let's now test the initial value of our Hook:

1. Define a new `test` function, passing the initial value `123` to the Hook:

```
test('should use initial value', () => {
    const { result } = renderHook(() => useCounter(123))
```

2. Now we can check if the `current` value equals the initial value, call `increment`, and ensure the `count` was increased from the initial value:

```
    expect(result.current.count).toBe(123)
    act(() => result.current.increment())
    expect(result.current.count).toBe(124) })
```

As we can see, we can simply pass the initial value to the Hook and check whether the value is the same.

Step 6: Testing reset and forcing re-rendering

We are now going to simulate the props of a component changing. Imagine the initial value for our Hook is a prop and it is initially `0`, which then changes to `123` later on. If we reset our counter now, it should reset to `123` and not `0`. However, to do so, we need to force the re-rendering of our test component after changing the value.

Let's now test resetting and forcing the component to re-render:

1. Define the `test` function and a variable for the `initial` value:

```
test('should reset to initial value', () => {
    let initial = 0
```

2. Next, we are going to render our Hook, but this time, we also pull out the `rerender` function via destructuring:

```
    const { result, rerender } = renderHook(() =>
useCounter(initial))
```

3. Now we set a new `initial` value and call the `rerender` function:

```
    initial = 123
    rerender()
```

4. Our `initial` value should now have changed, so when we call `reset`, the `count` will be set to `123`:

```
    act(() => result.current.reset())
    expect(result.current.count).toBe(123) })
```

As we can see, the testing library creates a dummy component, which is used for testing the Hook. We can force this dummy component to re-render in order to simulate what would happen when props change in a real component.

Exercise 12: Testing Context Hooks

Using the React Hooks Testing Library, we can also test more complex Hooks, such as Hooks making use of React context. Most of the custom Hooks we created for our blog app make use of contexts, so we are now going to test those. To test Hooks that use context, we first have to create a context wrapper, and then we can test the Hook.

In this section, we are going to perform the following:

- Create a `ThemeContextWrapper` component
- Test the `useTheme` Hook
- Create a `StateContextWrapper` component
- Test the `useDispatch` Hook
- Test the `useUserState` Hook
- Test the `usePostsState` Hook

Let's get started.

Step 1: Creating the ThemeContextWrapper

To be able to test the Theme Hook, we first have to set up the context and provide a wrapper component for the Hook's test component.

Let's now create the `ThemeContextWrapper` component:

1. Create a new `src/hooks/testUtils.js` file.
2. Import `React` and the `ThemeContext`, as follows:

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

3. Define a new function component called `ThemeContextWrapper`; it will accept `children` as props:

```
    export function ThemeContextWrapper ({ children }) {
```

`children` is a special prop of React components. It will contain all other components passed to it as `children`; for example, `<ThemeContextWrapper>{children}</ThemeContextWrapper>`.

4. We return a `ThemeContext.Provider` with our default theme, and then pass `children` to it:

```
    return (
        <ThemeContext.Provider value={{ primaryColor:
'deepskyblue', secondaryColor: 'coral' }}>
            {children}
        </ThemeContext.Provider>
    )
}
```

As we can see, a context wrapper simply returns a context provider component.

Step 2: Testing the useTheme Hook

Now that we have defined the `ThemeContextWrapper` component, we can make use of it while testing the `useTheme` Hook.
Let's now test the `useTheme` Hook as outlined in the following steps:

1. Create a new `src/hooks/useTheme.test.js` file.
2. Import the `renderHook` function as well as the `ThemeContextWrapper` and the `useTheme` Hook:

```
import { renderHook } from '@testing-library/react-hooks'
import { ThemeContextWrapper } from './testUtils'
import useTheme from './useTheme'
```

3. Next, define the `test` using the `renderHook` function and pass the `wrapper` as a second argument to it. Doing this will wrap the test component with the defined `wrapper` component, which means that we will be able to use the provided context in the Hook:

```
test('should use theme', () => {      const { result
} = renderHook(
        () => useTheme(),
        { wrapper: ThemeContextWrapper }
    )
```

4. Now we can check the result of our Hook, which should contain the colors defined in the `ThemeContextWrapper`:

```
    expect(result.current.primaryColor).toBe('deepskyblue')
    expect(result.current.secondaryColor).toBe('coral')
```

As we can see, after providing the context wrapper, we can test Hooks that use context just like we tested our simple Counter Hook.

Step 3: Creating the StateContextWrapper

For the other Hooks, which make use of the `StateContext`, we have to define another wrapper to provide the `StateContext` to the Hooks.

Let's now define the `StateContextWrapper` component with the following steps:

1. Edit `src/hooks/testUtils.js` and adjust the import statements to import the `useReducer` Hook, the `StateContext`, and the `appReducer` function:

```
import React, { useReducer } from 'react'
import { StateContext, ThemeContext } from '../contexts'
import appReducer from '../reducers'
```

2. Define a new function component called `StateContextWrapper`. Here we are going to use the `useReducer` Hook to define the app state, which is similar to what we did in the `src/App.js` file:

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

3. Next, define and return the `StateContext.Provider`, which is similar to what we did for the `ThemeContextWrapper`:

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

As we can see, creating a context wrapper always works similarly. However, this time, we are also defining a Reducer Hook in our wrapper component.

Step 4: Testing the useDispatch Hook

Now that we have defined the `StateContextWrapper`, we can use it to test the `useDispatch` Hook.

Let's test the `useDispatch` Hook with the following steps:

1. Create a new `src/hooks/useDispatch.test.js` file.
2. Import the `renderHook` function, the `StateContextWrapper` component, and the `useDispatch` Hook:

```
import { renderHook } from '@testing-library/react-hooks'
import { StateContextWrapper } from './testUtils'
import useDispatch from './useDispatch'
```

3. Then, define the `test` function, passing the `StateContextWrapper` component to it:

```
test('should use dispatch', () => {
    const { result } = renderHook(
        () => useDispatch(),
        { wrapper: StateContextWrapper }
    )
```

4. Finally, check whether the result of the Dispatch Hook is a function (the `dispatch` function):

```
    expect(typeof result.current).toBe('function') })
```

As we can see, using a `wrapper` component always works the same way, even if we use other Hooks within the `wrapper` component.

Step 5: Testing the useUserState Hook

Using the `StateContextWrapper` and the Dispatch Hook, we can now test the `useUserState` Hook by dispatching `LOGIN` and `REGISTER` actions and checking the result.
To dispatch these actions, we use the `act` function from the testing library.

Let's test the `useUserState` Hook:

1. Create a new `src/hooks/useUserState.test.js` file.
2. Import the necessary functions, the `useDispatch` and `useUserState` Hooks, and the `StateContextWrapper`:

```
import { renderHook, act } from '@testing-library/react-hooks'
import { StateContextWrapper } from './testUtils'
import useDispatch from './useDispatch'
import useUserState from './useUserState'
```

3. Next, we write a `test` that checks the initial `user` state:

```
test('should use user state', () => {
    const { result } = renderHook(
        () => useUserState(),
        { wrapper: StateContextWrapper }
    )
    expect(result.current).toBe('') })
```

4. Then, we write a `test` that dispatches a `LOGIN` action and then checks the new state. Instead of returning a single Hook, we now return an object with the results of both Hooks:

```
test('should update user state on login', () => {
    const { result } = renderHook(
        () => ({ state: useUserState(), dispatch: useDispatch() }),
        { wrapper: StateContextWrapper }
    )

    act(() => result.current.dispatch({ type: 'LOGIN', username:'Test
User' }))
    expect(result.current.state).toBe('Test User') })
```

5. Finally, we write a `test` that dispatches a `REGISTER` action and then checks the new state:

```
test('should update user state on register', () => {
    const { result } = renderHook(
        () => ({ state: useUserState(), dispatch: useDispatch() }),
        { wrapper: StateContextWrapper }
    )

    act(() => result.current.dispatch({ type: 'REGISTER', username:'Test
User' }))
    expect(result.current.state).toBe('Test User') })
```

As we can see, we can access both the `state` object and the `dispatch` function from our tests.

Step 6: Testing the usePostsState Hook

Similarly to how we tested the `useUserState` Hook, we can also test the `usePostsState` Hook.

Let's test the `usePostsState` Hook now:

1. Create a new `src/hooks/usePostsState.test.js` file.
2. Import the necessary functions, the `useDispatch` and `usePostsState` Hooks, and the `StateContextWrapper`:

```
import { renderHook, act } from '@testing-library/react-hooks'
import { StateContextWrapper } from './testUtils'
import useDispatch from './useDispatch'
import usePostsState from './usePostsState'
```

3. Then, we `test` the initial state of the `posts` array:

```
test('should use posts state', () => {
    const { result } = renderHook(
        () => usePostsState(),
        { wrapper: StateContextWrapper }
    )

    expect(result.current).toEqual([]) })
```

4. Next, we `test` whether a `FETCH_POSTS` action replaces the current `posts` array:

```
test('should update posts state on fetch action', () => {
    const { result } = renderHook(
        () => ({ state: usePostsState(), dispatch: useDispatch()
}),
        { wrapper: StateContextWrapper }
    )

    const samplePosts = [{ id: 'test' }, { id: 'test2' }]
    act(() => result.current.dispatch({ type: 'FETCH_POSTS', posts:
samplePosts }))
    expect(result.current.state).toEqual(samplePosts) })
```

5. Finally, we `test` whether a new post gets inserted on a `CREATE_POST` action:

```
test('should update posts state on insert action', () => {
    const { result } = renderHook(
        () => ({ state: usePostsState(), dispatch: useDispatch()
}),
        { wrapper: StateContextWrapper }
    )

    const post = { title: 'Hello World', content: 'This is a test',
author: 'Test User' }
    act(() => result.current.dispatch({ type: 'CREATE_POST',
...post }))
    expect(result.current.state[0]).toEqual(post) })
```

As we can see, the tests for the `posts` state are similar to the `user` state, but with different actions being dispatched.

Step 7: Testing async Hooks

Sometimes, we need to test Hooks that do asynchronous actions. This means that we need to wait a certain period of time until we check the result. To implement tests for these kind of Hooks, we can use the `waitForNextUpdate` function from the React Hooks Testing Library.

Before we can test async Hooks, we need to learn about the new JavaScript construct called `async/await`.

The async/await construct

Normal functions are defined as follows:

```
function doSomething () {
    // ...
}
```

Normal anonymous functions are defined as follows:

```
() => {
    // ...
}
```

Asynchronous functions are defined by adding the `async` keyword:

```
async function doSomething () {
    // ...
}
```

We can also make anonymous functions asynchronous:

```
async () => {
    // ...
}
```

Within `async` functions, we can use the `await` keyword to resolve promises. We do not have to do the following anymore:

```
() => {
    fetchAPITodos()
        .then(todos => dispatch({ type: FETCH_TODOS, todos }))
}
```

Instead, we can now do this:

```
async () => {
    const todos = await fetchAPITodos()
    dispatch({ type: FETCH_TODOS, todos }) }
```

As we can see, `async` functions make our code much more concise and easier to read! Now that we have learned about the `async/await` construct, we can start testing the `useDebouncedUndo` Hook.

Step 8: Testing the useDebouncedUndo Hook

We are going to use the `waitForNextUpdate` function to test debouncing in our `useDebouncedUndo` Hook by following these steps:

1. Create a new `src/hooks/useDebouncedUndo.test.js` file.
2. Import the `renderHook` and `act` functions as well as the `useDebouncedUndo` Hook:

```
import { renderHook, act } from '@testing-library/react-hooks'

import useDebouncedUndo from './useDebouncedUndo'
```

3. First of all, we `test` whether the Hook returns a proper `result`, including the `content` value, `setter` function, and the `undoRest` object:

```
test('should use debounced undo', () => {
    const { result } = renderHook(() => useDebouncedUndo())
    const [ content, setter, undoRest ] = result.current

    expect(content).toBe('')
    expect(typeof setter).toBe('function')
    expect(typeof undoRest.undo).toBe('function')
     expect(typeof undoRest.redo).toBe('function')
    expect(undoRest.canUndo).toBe(false)
    expect(undoRest.canRedo).toBe(false)
})
```

4. Next, we `test` whether the `content` value gets updated immediately:

```
test('should update content immediately', () => {
    const { result } = renderHook(() => useDebouncedUndo())
    const [ content, setter ] = result.current

    expect(content).toBe('')
    act(() => setter('test'))
    const [ newContent ] = result.current
    expect(newContent).toBe('test') })
```

   Remember that we can give any name to variables we pull out from an array using destructuring. In this case, we first name the `content` variable as `content`, then, later, we name it `newContent`.

5. Finally, we use `waitForNextUpdate` to wait for the debounced effect to trigger. After debouncing, we should now be able to undo our change:

```
test('should debounce undo history update', async () => {
    const { result, waitForNextUpdate } = renderHook(() =>
useDebouncedUndo())
    const [ , setter ] = result.current     act(()

=> setter('test'))

    const [ , , undoRest ] = result.current

    expect(undoRest.canUndo).toBe(false)

    await act(async () => await waitForNextUpdate())

    const [ , , newUndoRest ] = result.current

    expect(newUndoRest.canUndo).toBe(true) })
```

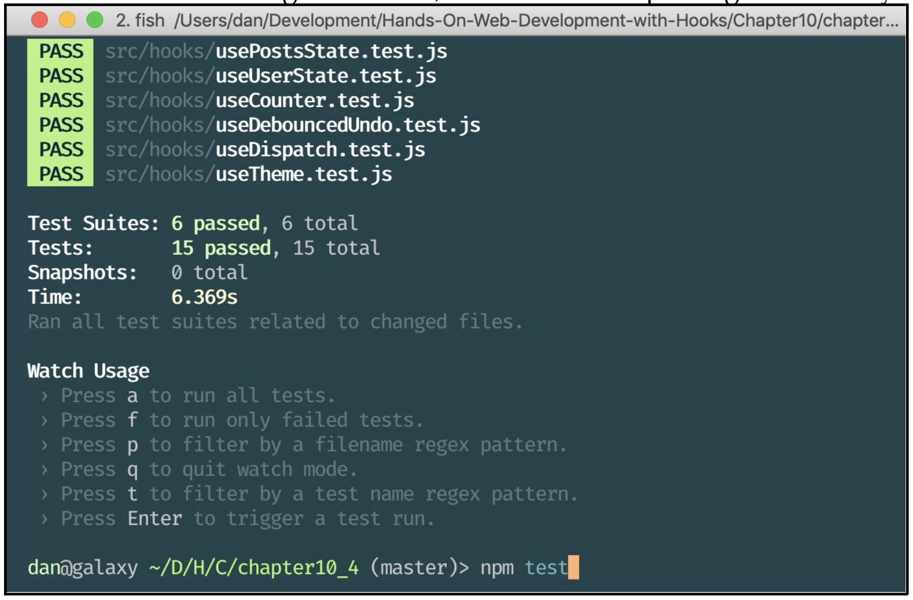
As we can see, we can use `async`/`await` in combination with the `waitForNextUpdate` function to easily handle testing asynchronous operations in our Hooks.

Step 9: Running the tests

To run the tests, simply execute the following command:

```
> npm test
```

As we can see from the following screenshot, all our tests are passing successfully:



```
2. fish  /Users/dan/Development/Hands-On-Web-Development-with-Hooks/Chapter10/chapter...
 PASS  src/hooks/usePostsState.test.js
 PASS  src/hooks/useUserState.test.js
 PASS  src/hooks/useCounter.test.js
 PASS  src/hooks/useDebouncedUndo.test.js
 PASS  src/hooks/useDispatch.test.js
 PASS  src/hooks/useTheme.test.js

Test Suites: 6 passed, 6 total
Tests:       15 passed, 15 total
Snapshots:   0 total
Time:        6.369s
Ran all test suites related to changed files.

Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press p to filter by a filename regex pattern.
 › Press q to quit watch mode.
 › Press t to filter by a test name regex pattern.
 › Press Enter to trigger a test run.

dan@galaxy ~/D/H/C/chapter10_4 (master)> npm test
```

All Hook tests passing successfully

The test suite actually watches for changes in our files and automatically reruns tests. We can use various commands to manually trigger test reruns and we can press *Q* to quit the test runner.

Exercise 13: Exploring the React Hooks API

The official React library provides certain built-in Hooks, which can be used to create custom Hooks. We have already learned about the three basic Hooks that React provides:

- `useState`
- `useEffect`
- `useContext`

Additionally, React provides more advanced Hooks, which can be very useful in certain use cases:

- `useReducer`
- `useCallback`
- `useMemo`
- `useRef`
- `useImperativeHandle`
- `useLayoutEffect`
- `useDebugValue`

The useState Hook

The `useState` Hook returns a value that will persist across re-renders, and a function to update it. A value for the `initialState` can be passed to it as an argument:

```
const [ state, setState ] = useState(initialState)
```

Calling `setState` updates the value and re-renders the component with the updated value. If the value did not change, React will not re-render the component.

A function can also be passed to the `setState` function, with the first argument being the current value. For example, consider the following code:

```
    setState(val => val + 1)
```

Furthermore, a function can be passed to the first argument of the Hook if the initial state is the result of a complex computation. In that case, the function will only be called once during the initialization of the Hook:

```
    const [ state, setState ] = useState(() => {
        return computeInitialState() })
```

The State Hook is the most basic and ubiquitous Hook provided by React.

The useEffect Hook

The `useEffect` Hook accepts a function that contains code with side effects, such as timers and subscriptions. The function passed to the Hook will run after the render is done and the component is on the screen:

```
    useEffect(() => {
        // do something
    })
```

A cleanup function can be returned from the Hook, which will be called when the component unmounts and is used to, for example, clean up timers or subscriptions:

```
useEffect(() => {
    const interval = setInterval(() => {}, 100)
    return () => {
        clearInterval(interval)
    }
})
```

The cleanup function will also be called before the effect is triggered again, when dependencies of the effect update.

To avoid triggering the effect on every re-render, we can specify an array of values as the second argument to the Hook. Only when any of these values change, the effect will get triggered again:

```
useEffect(() => {
    // do something when state changes
}, [state])
```

This array passed as the second argument is called the dependency array of the effect. If you want the effect to only trigger during mounting, and the cleanup function during unmounting, we can pass an empty array as the second argument.

The useContext Hook

The `useContext` Hook accepts a context object and returns the current `value` for the context. When the context provider updates its `value`, the Hook will trigger a re-render with the latest `value`:

```
const value = useContext(NameOfTheContext)
```

It is important to note that the context object itself needs to be passed to the Hook, not the consumer or provider.

The useReducer Hook

The `useReducer` Hook is an advanced version of the `useState` Hook. It accepts a `reducer` as the first argument, which is a function with two arguments: `state` and `action`. The `reducer` function then returns the updated state computed from the current state and the action. If a reducer returns the same value as the previous state, React will not re-render components or trigger effects:

```
const [ state, dispatch ] = useReducer(reducer, initialState, initFn)
```

We should use the `useReducer` Hook instead of the `useState` Hook when dealing with complex `state` changes. Furthermore, it is easier to deal with global `state` because we can simply pass down the `dispatch` function instead of multiple setter functions.

The `dispatch` function is stable and will not change on re-renders, so it is safe to omit it from `useEffect` or the `useCallback` dependencies

We can specify the initial `state` by setting the `initialState` value or specifying an `initFn` function as the third argument. Specifying such a function makes sense when computing the initial `state` takes a long time or when we want to reuse the function to reset `state` through an `action`.

The useMemo Hook

The `useMemo` Hook takes a result of a function and memoizes it. This means that it will not be recomputed every time. This Hook can be used for performance optimizations:

```
const memoizedVal = useMemo(     ()
=> computeVal(a, b, c),
    [a, b, c]
)
```

In the previous example, `computeVal` is a performance-heavy function that computes a result from `a`, `b`, and `c`.

> `useMemo` runs during rendering, so make sure the computation function does not cause any side effects, such as resource requests. Side effects should be put into a `useEffect` Hook.

The array passed as the second argument specifies the dependencies of the function. If any of these values change, the function will be recomputed; otherwise, the stored result will be used. If no array is provided, a new value will be computed on every render. If an empty array is passed, the value will only be computed once.

> Do not rely on `useMemo` to only compute things once. React may forget some previously memoized values if they are not used for a long time, for example, to free up memory. Use it only for performance optimizations.

The `useMemo` Hook is used for performance optimizations in React components.

The useCallback Hook

The `useCallback` Hook works similarly to the `useMemo` Hook. However, it returns a memoized callback function instead of a value:

```
const memoizedCallback = useCallback(
    () => doSomething(a, b, c),
    [a, b, c]
)
```

The previous code is similar to the following `useMemo` Hook:

```
const memoizedCallback = useMemo(
    () => () => doSomething(a, b, c),
    [a, b, c]
)
```

The function returned will only be redefined if one of the dependency values passed in the array of the second argument changes.

The useRef Hook

The `useRef` Hook returns a ref object that can be assigned to a component or element via the `ref` prop. Refs can be used to deal with references to elements and components in React:

```
const refContainer = useRef(initialValue)
```

After assigning the ref to an element or component, the ref can be accessed via `refContainer.current`. If `InitialValue` is set, `refContainer.current` will be set to this value before assignment.

The following example defines an `input` field that will automatically be focused when rendered:

```
function AutoFocusField () {
    const inputRef = useRef(null)
    useEffect(() => inputRef.current.focus(), [])
    return <input ref={inputRef} type="text" /> }
```

It is important to note that mutating the current value of a ref does not cause a re-render. If this is needed, we should use a `ref` callback using `useCallback` as follows:

```
function WidthMeasure () {
    const [ width, setWidth ] = useState(0)

    const measureRef = useCallback(node => {
        if (node !== null) {
            setWidth(node.getBoundingClientRect().width)
        }
    }, [])

    return <div ref={measureRef}>I am {Math.round(width)}px wide</div>
}
```

Refs can be used to access the DOM, but also to keep mutable values around, such as storing references to intervals:

```
function Timer () {
    const intervalRef = useRef(null)

    useEffect(() => {
        intervalRef.current = setInterval(doSomething, 100)
return () => clearInterval(intervalRef.current)

    })

    // ...
}
```

Using refs like in the previous example makes them similar to instance variables in classes, such as `this.intervalRef`.

The useImperativeHandle Hook

The `useImperativeHandle` Hook can be used to customize instance values that are exposed to other components when pointing a `ref` to it. Doing this should be avoided as much as possible, however, as it tightly couples components together, which harms reusability.

The `useImperativeHandle` Hook has the following signature: `useImperativeHandle(ref,`

```
createHandle, [dependencies])
```

We can use this Hook to, for example, expose a `focus` function that other components can trigger via a `ref` to the component. This Hook should be used in combination with `forwardRef` as follows:

```
function FocusableInput (props, ref) {
    const inputRef = useRef()
    useImperativeHandle(ref, () => ({
        focus: () => inputRef.current.focus()
    }))
    return <input {...props} ref={inputRef} />
}
FocusableInput = forwardRef(FocusableInput)
```

Then, we can access the `focus` function as follows:

```
function AutoFocus () {
     const inputRef = useRef()
    useEffect(() => inputRef.current.focus(), [])

    return <FocusableInput ref={inputRef} /> }
```

As we can see, using refs means that we can directly access elements and components.

The useLayoutEffect Hook

The `useLayoutEffect` Hook is identical to the `useEffect` Hook, but it fires synchronously after all DOM mutations are completed and before the component is rendered in the browser. It can be used to read information from the DOM and adjust the appearance of components before rendering. Updates inside this Hook will be processed synchronously before the browser renders the component.

Do not use this Hook unless it is really needed, which is only in certain edge cases. `useLayoutEffect` will block visual updates in the browser, and thus, is slower than `useEffect`.

The rule here is to use `useEffect` first. If your mutation changes the appearance of the DOM node, which can cause it to flicker, you should use `useLayoutEffect` instead.

The useDebugValue Hook

The `useDebugValue` Hook is useful for developing custom Hooks that are part of shared libraries. It can be used to show certain values for debugging in React DevTools.

For example, in our `useDebouncedUndo` custom Hook, we could do the following:

```
export default function useDebouncedUndo (timeout = 200) {
    const [ content, setInput ] = useState('')
    const [ undoContent, { set: setContent, ...undoRest } ] = useUndo('')

useDebugValue('init')

    const [ setDebounce, cancelDebounce ] = useDebouncedCallback(
        (value) => {
            setContent(value)
            useDebugValue('added to history')
        },
        timeout
    )
    useEffect(() => {
        cancelDebounce()
        setInput(undoContent.present)
        useDebugValue(`waiting ${timeout}ms`)
    }, [cancelDebounce, undoContent])
    function setter (value) {
        setInput(value)
        setDebounce(value)
    }
    return [ content, setter, undoRest ] }
```

Adding these `useDebugValue` Hooks will show the following in the React DevTools:

- When the Hook is initialized: **DebouncedUndo: init**
- When a value was entered: **DebouncedUndo: waiting 200 ms**
- After debouncing (after `200` ms): **DebouncedUndo: added to history**