Step 4: Counters

# Intruduction:

New features in this exercise:

* [useEffect hook](https://reactjs.org/docs/hooks-effect.html)
* [setTimeout method](https://www.w3schools.com/jsref/met_win_settimeout.asp)

In this exercise we’re going to take a look at React **useEffect hook**. To better demonstrate the hook and how to trigger it, we will also learn a bit about JavaScript **setTimeout method**. We will create 3 separate “counters” that will show the difference between the ways to trigger it.

Start by creating a new file called **Counters.jsx** and creating a base for your component. **Import** React, useState and **useEffect**.

import React from "react";

import { useState, useEffect } from "react";

export const Counters = () =>{

    return (

        <div>

        </div>

    )

}

**Import the component to App.js, add a route for the component and a link to Router.jsx so we can see our progress during the exercise:**

App.js

function App() {

    return (

        <BrowserRouter>

            <div>

                <Routes>

                    <Route path='/' element={<Router />}>

                        <Route index element={<Message />} />

                        <Route path='Buttons' element={<Buttons />} />

                        <Route path='Counters' element={<Counters />} />

                    </Route>

                </Routes>

            </div>

        </BrowserRouter>

    );

}

Router.jsx

export const Router = () => {

  return (

    <>

        <div>

            <ul>

                <li>

                    <Link to="/">Home </Link>

                </li>

                <li>

                    <Link to="/buttons">Buttons</Link>

                </li>

                <li>

                    <Link to="/counters">Counters</Link>

                </li>

            </ul>

        </div>

        <Outlet />

    </>

  )

};

# useEffect:

We will use the useEffect hook in this exercise and learn about different ways of triggering it.

What is useEffect?

The useEffect hook lets you perform **side effects** in function components. The side effects can be anything from data fetching or directly updating the DOM to setting up timers or intervals, you name it. Your component can have multiple useEffect hooks at the same time. By default, useEffect will run when the component is first created and from there on, every render after that, unless a **dependency array** is added.

There are several ways to trigger the hook, by default it will **run on every render**. But this can **be bad practice** and is not very useful in development, we will show what will happen later in the exercise. Below is an example of a hook without a dependency array.

useEffect(() => {

    console.log("I will run on every render");

});

To get the hook to only run **once**, we will add a dependency array. It is simply adding square brackets at the end of the hook like shown below.

useEffect(() => {

    console.log("I will run only on the initial render");

}, []);

The third way to run it, is by adding a dependency **to the** array. This can be a state for an example. Let’s say that the dependency is a state called “count”. So now the hook will run on the initial render **and** every time the state is changed (setCount). Example of this below:

useEffect(() => {

    console.log("I will run on the initial render and every time the count state is changed");

}, [count]);

You can pass it multiple dependencies, it will trigger when **any of them** change.

# setTimeout:

To learn the third way to trigger useEffect, we will take a look at JavaScript **setTimeout method**.

What is setTimeout?

setTimeout method is used to **set a timer** to a function **before** it runs. Inside the parentheses you will first input the name of the function you want to have a delay on and a time after that.

setTimeout(function, 1000);

**The time will be displayed in milliseconds**(1 second = 1000 milliseconds).

Normally the method is only executed once. In JavaScript, if you need repeated executions, you would use [setInterval method](https://www.w3schools.com/jsref/met_win_setinterval.asp). But here we will create multiple executions with the setTimeout method by using the useEffect hook to better understand the different triggers.

# Counters:

useEffect on initial render:

Let’s start writing our component now. First, we will create a counter that’s going to only run on the **initial render**. We will use the useState hook for all the counters, start by creating a state called count, with an **initial state of 0**. Create an empty useEffect hook:

useEffect(() => {

});

Inside the hook, add the setTimeout method. This will be a bit trickier than in the example since we’re using useState inside. You will set the timeout for our setCount function using the statement below.

setTimeout(function () {

    setCount(count + 1)

}, 1000);

Let’s add the dependency array now. Just like in the example we looked at before, **square brackets** at the end of the hook:

useEffect(() => {

    setTimeout(function () {

        setCount(count + 1)

    }, 1000);

}, []);

Now we will render our count. Inside the div element, add a header to render the count:

return (

    <div>

        <h1>I've rendered {count} time!</h1>

    </div>

)

Move to your browser and see if the count is visible. Because we added the empty dependency array, the counter will only run **once** and stop at 1. Your page should look something like this:

Graphical user interface, text, application

Description automatically generated

useEffect without a depency array and with setTimeout:

Next, we will create a counter that will add one to our state **every second**, like a timer. It will be almost identical to our first one. Create state to this second counter, simply name it count2.

At this point we will take a look at what happens when you make a counter **without a** dependency array or the setTimeout method. This will visualize why it’s **not** a good idea to use it like this.

Create an useEffect hook without the delay. Inside set the count2 state to add one to the initial state at every render:

useEffect(() => {

    setCount2(count2 + 1);

});

And render the count2 inside the div.

return (

    <div>

        <h1>I've rendered {count} time!</h1>

        <h1>I've rendered {count2} times!</h1>

    </div>

)

Open your browser and you will see what happens.

Graphical user interface, text, application, chat or text message, email

Description automatically generated

This is happening because our hook will run on the initial run, **adding one** to the count2 state which will trigger the hook again, **instantly**. So the hook will run as fast as it can and you **shouldn’t** use it like this.

Let’s **change** the hook to save our eyes. We will add the setTimeout method like in the previous counter, this time we will not add the dependency array:

useEffect(() => {

    setTimeout(function () {

        setCount2(count2 + 1)

    }, 1000);

});

Now let’s move to the browser again and you will see that it’s rendering every second, this is because the setTimeout method is saving us by **delaying** the code from running by 1 second every time the hook is triggered again.

Graphical user interface, text, email

Description automatically generated

useEffect with dependency:

Now let’s add the third, and final counter, create your third state (count3) initial state same as the previous one. Here we will use our count2 state as the hooks **trigger**. Create your hook:

useEffect(() => {

});

Inside, add your setCount3 to add one at every render. Now, if we would render the count, the same exact thing would happen. But this time we will add a **dependency** inside the array. The dependency being our **previous state**, which runs every second and changes the state. That change will trigger our hook to run at **exactly same time** as our previous counter:

useEffect(() => {

    setCount3(count3 +1 );

}, [count2]);

Add the count3 state to our div and open the browser:

Graphical user interface, text, application

Description automatically generated

You will see the counter updating at the same phase as our count2. You might be confused as why the counter **starts at 1**, the reason for that is that the third counter is lacking the delay our count2 has. So the hook will run instantly **once**(since the useEffect will run every time at least once). And after that it will only react to the count2 state changing, which is every second.

This is the end of this exercise. Hopefully you learned how useEffect works and the difference in its dependencies. In the next exercise we will learn to render items from an external file using props and mapping. **See you in there!** 😊