React Functional Components: State, Props, and Lifecycle Methods

Introduction

Components in React are the building blocks of an application. Basically, a React component returns a JSX element that is rendered in the UI. Moreover, a component can have other aspects such as state, props, and lifecycle methods.

React has two types of components:

- Class components
- Functional components

As the name suggests, a class component is a JavaScript class extended to a React Component. It has a mandatory `render()` method which returns a JSX element. Earlier, React was mostly class-based because "state" was only supported in class components. But in React 16.8, React hooks were introduced and these hooks changed everything in React development.

With React hooks, "state" could be declared in functional components, thus, transforming them into stateful components from stateless components.

Today, most professional React developers prefer and recommend functional components over class components. Thus making functional components, present as well as the future of React development. In this article, we will discuss what are the functional components in React.

Why functional components?

A functional component is a plain JavaScript function that returns a JSX element.

```
function SayHello(){
  return(
    <h2>Hello World!</h2>
  )
}
export default SayHello;
```

In the above code, SayHello is a functional component and it is returning a JSX element. In the end, the SayHello function is being exported. Basically, SayHello is nothing but a plain JavaScript function.

Now, observe the following code.

```
import React from "react";
class SayHello extends React.Component {
  render() {
    return <h2>Hello World!</h2>;
  }
}
export default SayHello;
```

The above class component is the equivalent of the earlier functional component. It is easily observable that the functional component is more easily readable than the class component. Moreover, it is easier to work with functional components because writing class components is more complex and complicated.

State in functional components

As mentioned earlier, it was not possible to use state in functional components because the setState() method was only supported in the class components. But with React hooks, now it is possible to use state in functional components.

The state of a component in React is a plain JavaScript object that controls the behavior of a component. The change in a state triggers component re-renders.

Two main React hooks are used to declare and manipulate the state in a function component.

- useState
- useReducer

The useState hook has the following syntax.

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

The useState hook declares a state variable that holds the state and a function (setState) that can update the state. Moreover, the state variable can be initialized by passing a value (initialState) to the useState hook.

Let’s understand with the help of an example.

```
import { useState } from "react";


const Counter = () => {
  const [count, setCounter] = useState(0);


  return (
    <>
      <button onClick={() => setCounter(count + 1)}> Increment by 1
</button>
      &nbsp;&nbsp;
      <button onClick={() => setCounter(count - 1)}> Decrement by 1
</button>
      <h2>{count}</h2>
    </>
  );
};


export default Counter;
```

In the above code, count is declared using the useState hook with 0 as its initial value.

```
const [count, setCounter] = useState(0);
```

The Counter component renders two buttons in the UI, first incrementing the value of count by 1 and the other decrementing the value of count by 1.

```
<button onClick={() => setCounter(count + 1)}> Increment by 1
</button>
```

```
<button onClick={() => setCounter(count - 1)}> Decrement by 1
</button>
```

To update the value of the state variable, we have to pass the new value to the function declared using the useState hook.

In the end, the state is rendered in the UI.

```
<h2>{count}</h2>
```

Let’s check the output in the browser.

The useReducer hook is used for complex state management. When the state is inter-dependable, the useReducer hook is preferred over the useState hook. To understand the useReducer, one should know what is a reducer and how to use it.

Basically, a reducer is a pure function that takes state and action as arguments and returns a new state.

Following is the syntax of the useReducer hook.

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

It is very similar to the useState hook but instead of creating a function to update the state, the useReducer hook creates a function for dispatching actions. The useReducer hook has two arguments - reducer and an initialState. There is a third optional argument for initializing the state lazily.

Let’s try to understand the useReducer hook with the help of an example. Observe the following component.

```
const Counter = () => {
  return (
    <>
      <button>Increment</button>
      &nbsp;
      <button>Decrement</button>
      <hr />
      <h3>Total: </h3>
      <h3>Total increment: </h3>
      <h3>Total decrement: </h3>
    </>
  );
};

export default Counter;
```

The Counter component has two buttons that will increment and decrement a value by 1. Along with the total value, the component will track the total increment as well as the total decrement. So, instead of having a single state variable, we will need three state variables, each for total value, total increment, and total decrement.

Observe the initial state.

```
const initialState = {
  total: 0,
  totalIncrement: 0,
  totalDecrement: 0,
};
```

Now, we need a reducer to return a new state.

```
const reducer = (state, action) => {
  switch (action.type) {
    case "INCREMENT":
      return {
        ...state,
        total: state.total + 1,
        increment: state.increment + 1,
      };
    case "DECREMENT":
      return {
        ...state,
        total: state.total - 1,
        decrement: state.decrement + 1,
      };
  }
};
```

Let’s use the useReducer hook.

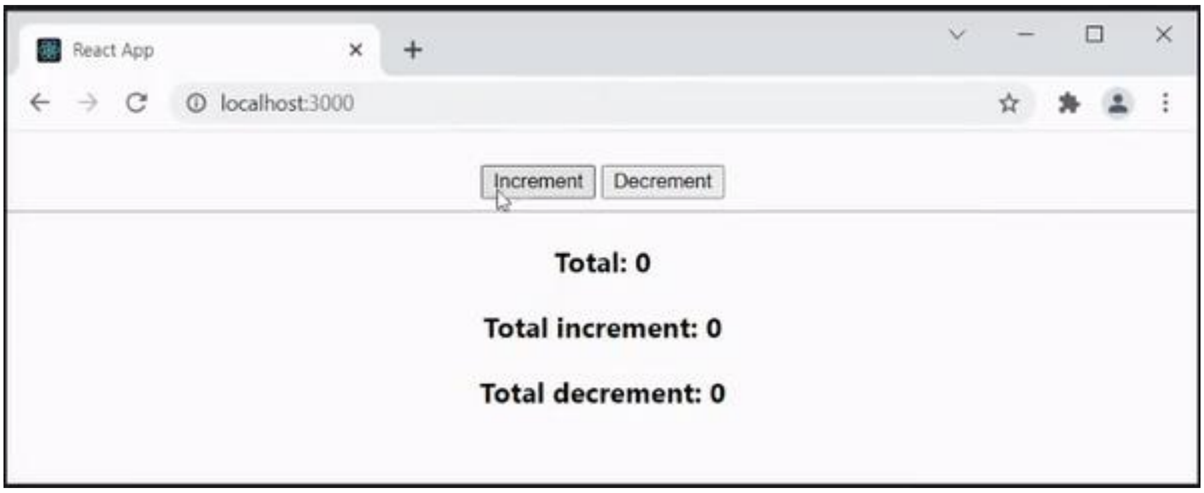
```
const [values, dispatch] = useReducer(reducer, initialState);
```

values will have the state while dispatch will be used to fire the actions.

Observe the return statement of the Counter component.

```
return (
  <>
    <button onClick={() => dispatch({ type: "INCREMENT"
})}>Increment</button>
    &nbsp;
    <button onClick={() => dispatch({ type: "DECREMENT"
})}>Decrement</button>
    <hr />
    <h3>Total: {values.total} </h3>
    <h3>Total increment: {values.totalIncrement} </h3>
    <h3>Total decrement: {values.totalDecrement} </h3>
  </>
);
```

Each button is dispatching an action and below them, the state values are being used. Let’s check the output.



Total: 0

Total increment: 0

Total decrement: 0

Props in functional components

While the state is managed within the component, props are passed from a parent component to a child component. Props are properties. In simple words, props are plain JavaScript objects but immutable.

Props in the functional components are almost similar to the class components. There is nothing special required for using props in functional components.

Observe the Employees component.

```
import Employee from "./Employee";

const Employees = () => {
  const data = [
    {
      name: "Johny",
      age: 21,
      city: "New York",
    },
    {
      name: "Mike",
      age: 27,
      city: "Detroit",
    },
    {
      name: "Sophie",
      age: 34,
      city: "Chicago",
    },
  ];

  return data.map((emp) => {
    return <Employee name={emp.name} age={emp.age} city={emp.city}
/>;
  });
};

export default Employees;
```

data is an array of objects and the properties of every object are being passed as props to the Employee component.

```
return data.map((emp) => {
  return <Employee name={emp.name} age={emp.age} city={emp.city} />;
});
```

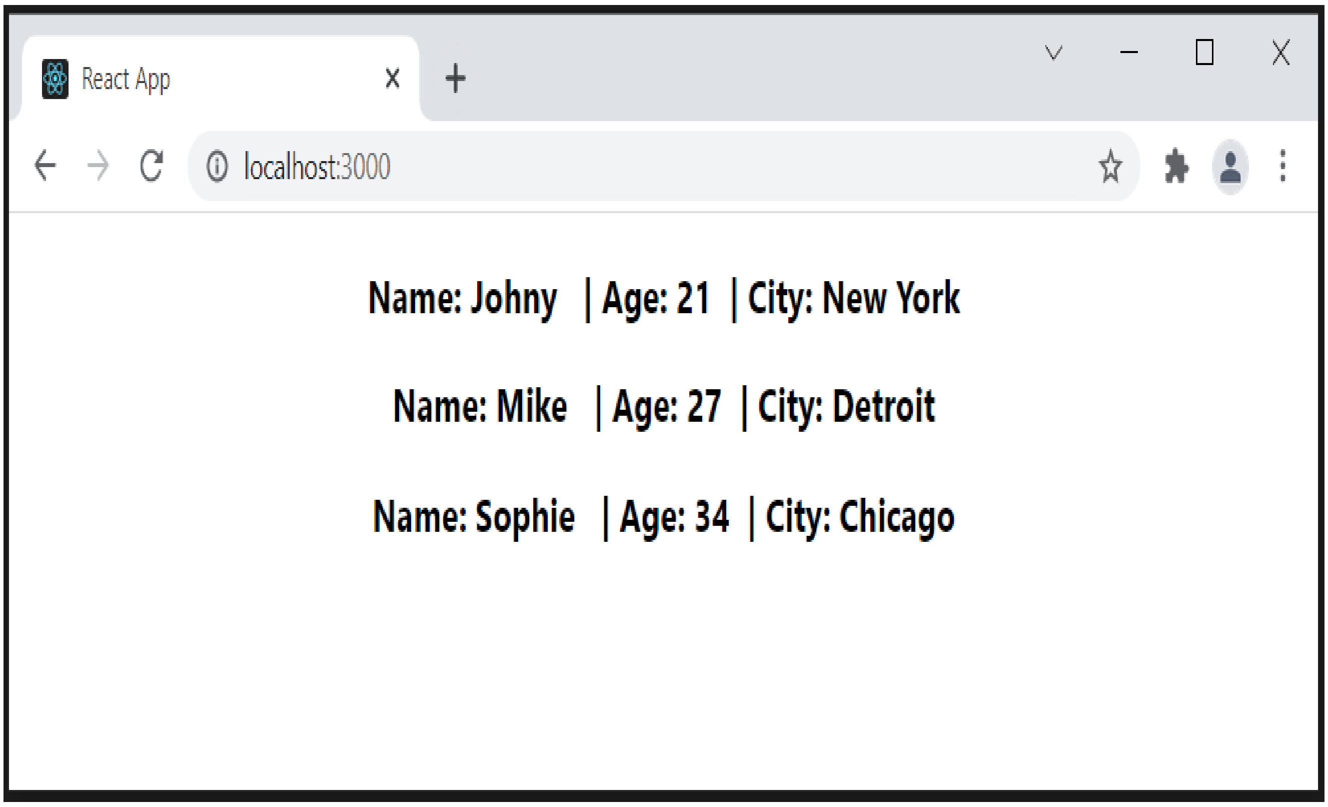
In the Employee component, these values are being rendered in the UI.

```
const Employee = (props) => {
  return (
    <>
      <h3>
        Name: {props.name}| &nbsp; Age: {props.age} |&nbsp; City:
{props.city}
      </h3>
    </>
  );
};

export default Employee;
```

To use the props in a functional component, it is mandatory to pass them as arguments. Moreover, we can also destruct them in functional components.

```
const Employee = ({ name, age, city }) => {
  return (
    <>
      <h3>
        Name: {name} &nbsp; | Age: {age} &nbsp;| City: {city}
      </h3>
    </>
  );
};

export default Employee;
```

The output will be the same.



```
Name: Johny  | Age: 21 | City: New York

Name: Mike  | Age: 27 | City: Detroit

Name: Sophie  | Age: 34 | City: Chicago
```

Lifecycle methods of functional components

Lifecycle methods are special in-built methods that are supported only in the class components. Each of these methods executes a specific phase during the lifecycle of a component.

Every React component has a lifecycle. These are mounting, updating, and unmounting. The lifecycle methods are very useful because sometimes, we may want to execute a piece of code at a specific time. For example, suppose we want to call an API just after the component is mounted. In such a case, we can use the `componentDidMount` lifecycle method.

React provides another special hook to create lifecycle methods like functionality in the functional components. This hook is called `useEffect`.

The `useEffect` hook is used to perform side-effects. It has the following syntax.

```
useEffect(() => {
  effect;
  return () => {
    cleanup;
  };
}, [input]);
```

It has two arguments - a callback function and a dependency array. So what happens is that the callback function executes according to the presence and value of the dependency array and this is how lifecycle methods like functionality are achieved in functional components.

componentDidMount

The `componentDidMount` lifecycle method executes once after initial rendering. If an empty dependency array is provided as the second argument, the callback function executes only after initial rendering, this behaving like `componentDidMount`.

componentDidUpdate

The componentDidUpdate lifecycle method executes whenever the component is re-render. So every time the state is changed, the componentDidUpdate method executes. If no dependency array is provided, the callback function will execute whenever the component re-renders, including the initial rendering.

If values are provided in the dependency array, the callback function will execute during initial rendering and whenever any of the values specified changes.

componentWillUnmount

The componentWillUnmount lifecycle method executes when the component is unmounted. If a function is returned from the call function, that function executes when the component is unmounted.

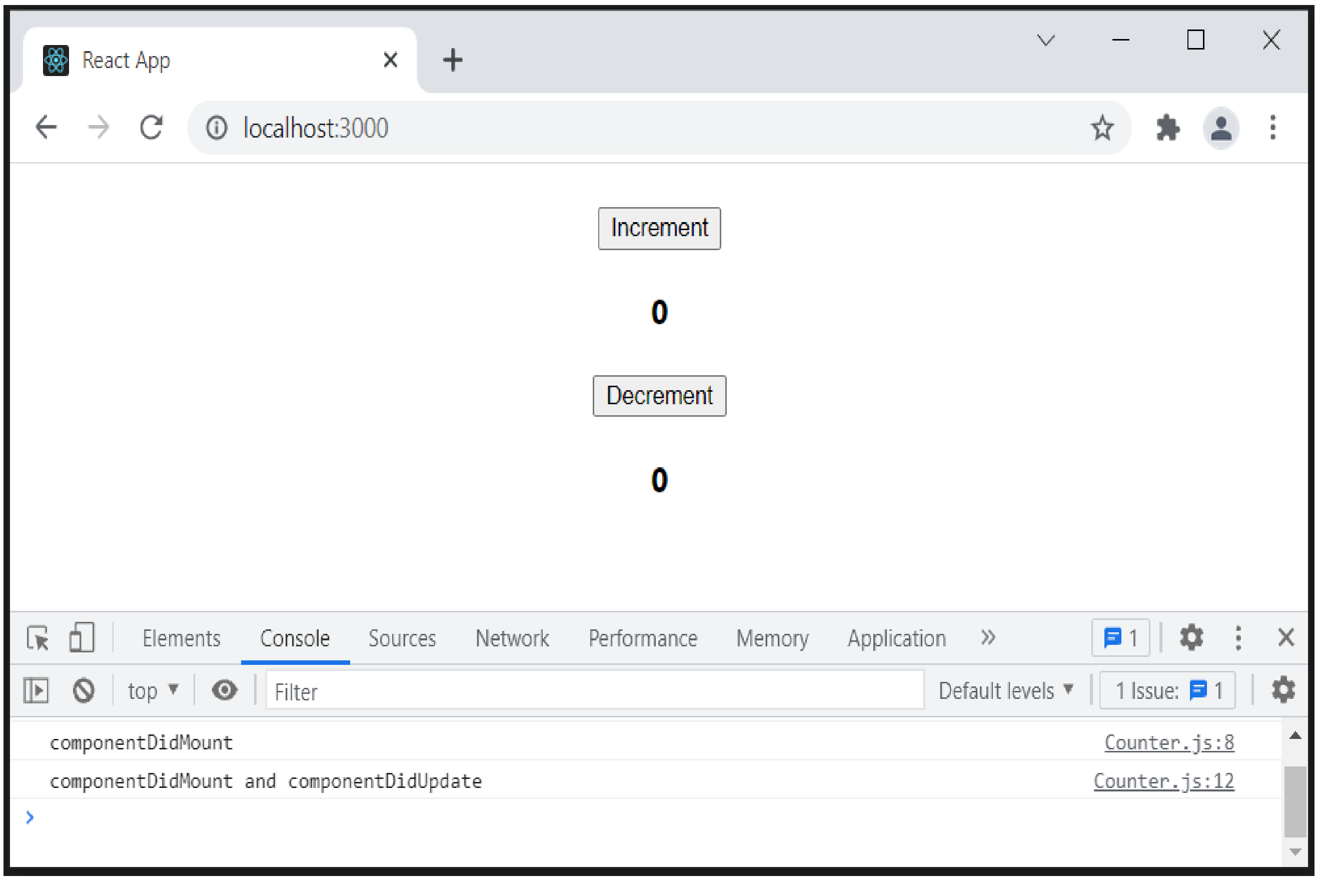
Let’s understand with the help of an example.

```
import { useState, useEffect } from "react";

const Counter = () => {
  const [increment, setIncrement] = useState(0);
  const [decrement, setDecrement] = useState(0);

  useEffect(() => {
    console.log("componentDidMount");
  }, []);

  useEffect(() => {
    console.log("componentDidMount and componentDidUpdate");
  });

  return (
    <>
      <button onClick={() => setIncrement(increment +
1)}>Increment</button>
      <h3>{increment}</h3>
      <button onClick={() => setDecrement(decrement -
1)}>Decrement</button>
      <h3>{decrement}</h3>
    </>
  );
};

export default Counter;
```

The above code has two states - increment and decrement. Observe the useEffect hooks used in the component.

```
useEffect(() => {
  console.log("componentDidMount");
}, []);

useEffect(() => {
  console.log("componentDidMount and componentDidUpdate");
});
```

The first useEffect will execute only after the initial rendering while the second one will be executed after the initial rendering as well as when the component re-renders.



```
componentDidMount                                  Counter.js:8
componentDidMount and componentDidUpdate           Counter.js:12
```

After the initial rendering, both the hooks are executed once. The second will execute whenever any button is clicked. This happens because both the buttons update the state and the useEffect has no dependency array.

Let’s make some changes to the code.

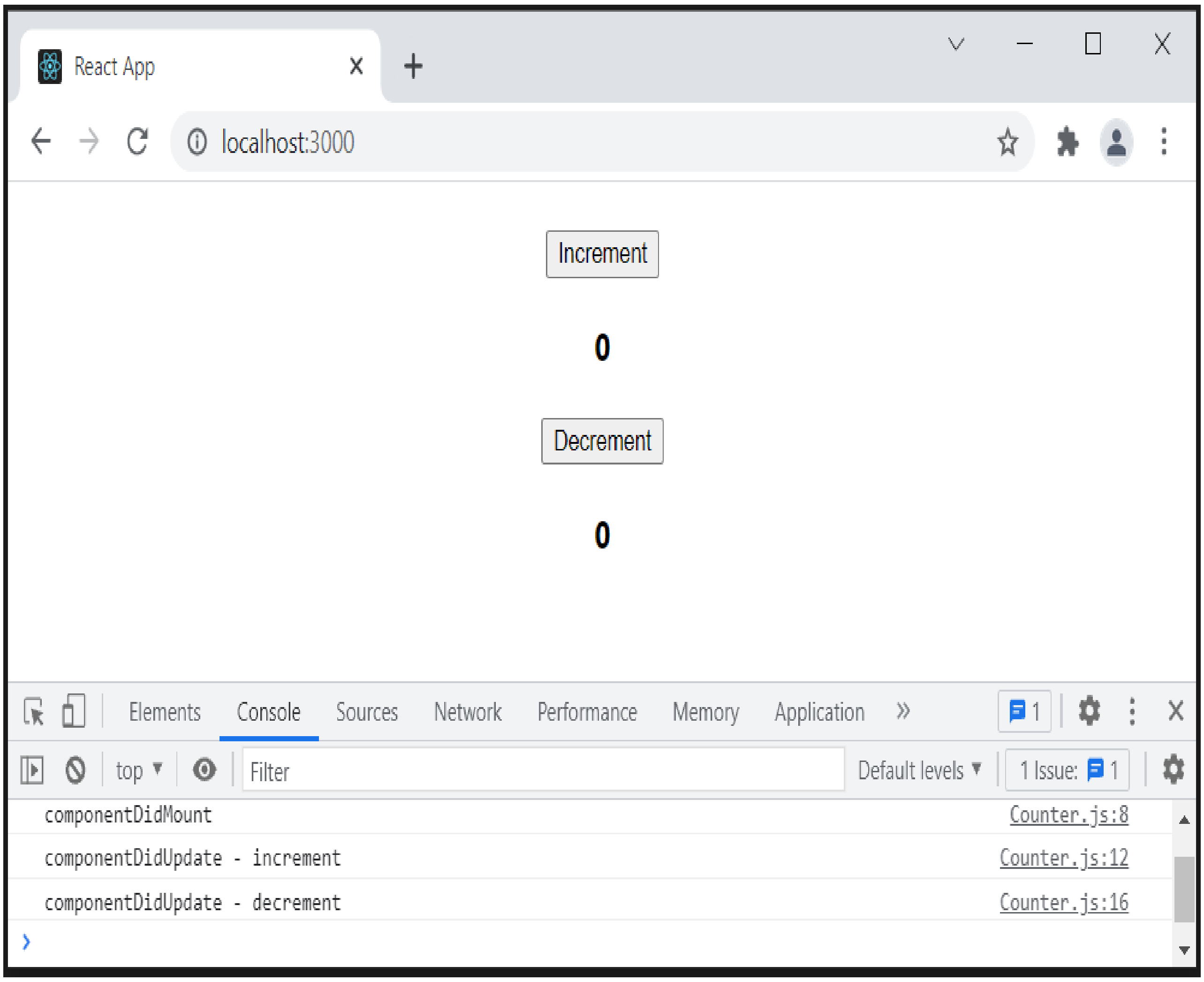
```
import { useState, useEffect } from "react";

const Counter = () => {
  const [increment, setIncrement] = useState(0);
  const [decrement, setDecrement] = useState(0);

  useEffect(() => {
    console.log("componentDidMount");
  }, []);

  useEffect(() => {
    console.log("componentDidUpdate - increment");
  }, [increment]);

  useEffect(() => {
    console.log("componentDidUpdate - decrement");
  }, [decrement]);

  return (
    <>
      <button onClick={() => setIncrement(increment + 
1)}>Increment</button>
      <h3>{increment}</h3>
      <button onClick={() => setDecrement(decrement - 
1)}>Decrement</button>
      <h3>{decrement}</h3>
    </>
  );
};

export default Counter;
```

Now, two new `useEffect` hooks are added.

```
useEffect(() => {
  console.log("componentDidUpdate - increment");
}, [increment]);

useEffect(() => {
  console.log("componentDidUpdate - decrement");
}, [decrement]);
```

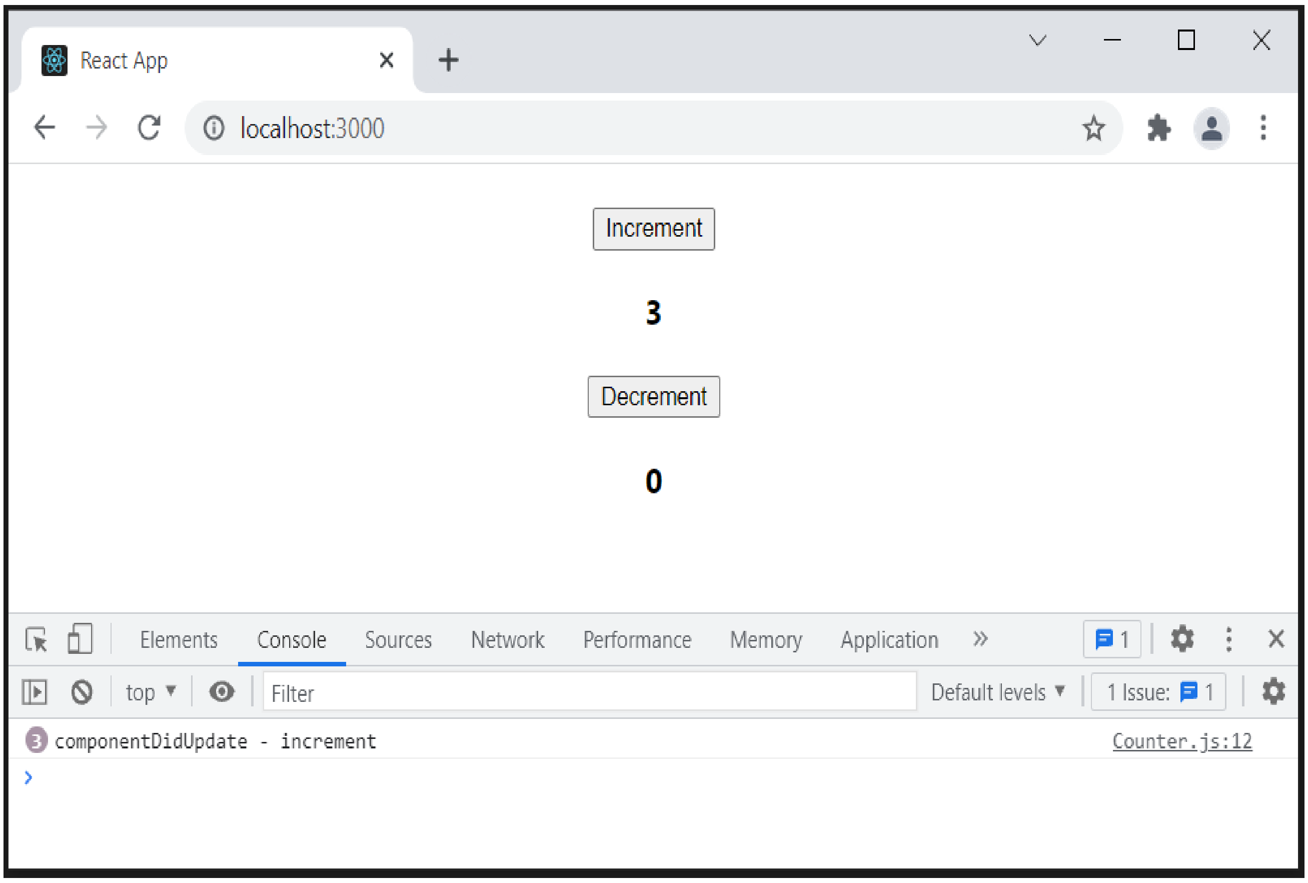
Both of them have arrays with values, meaning, they will execute during initial rendering and when the provided values are changed.

After initial rendering:



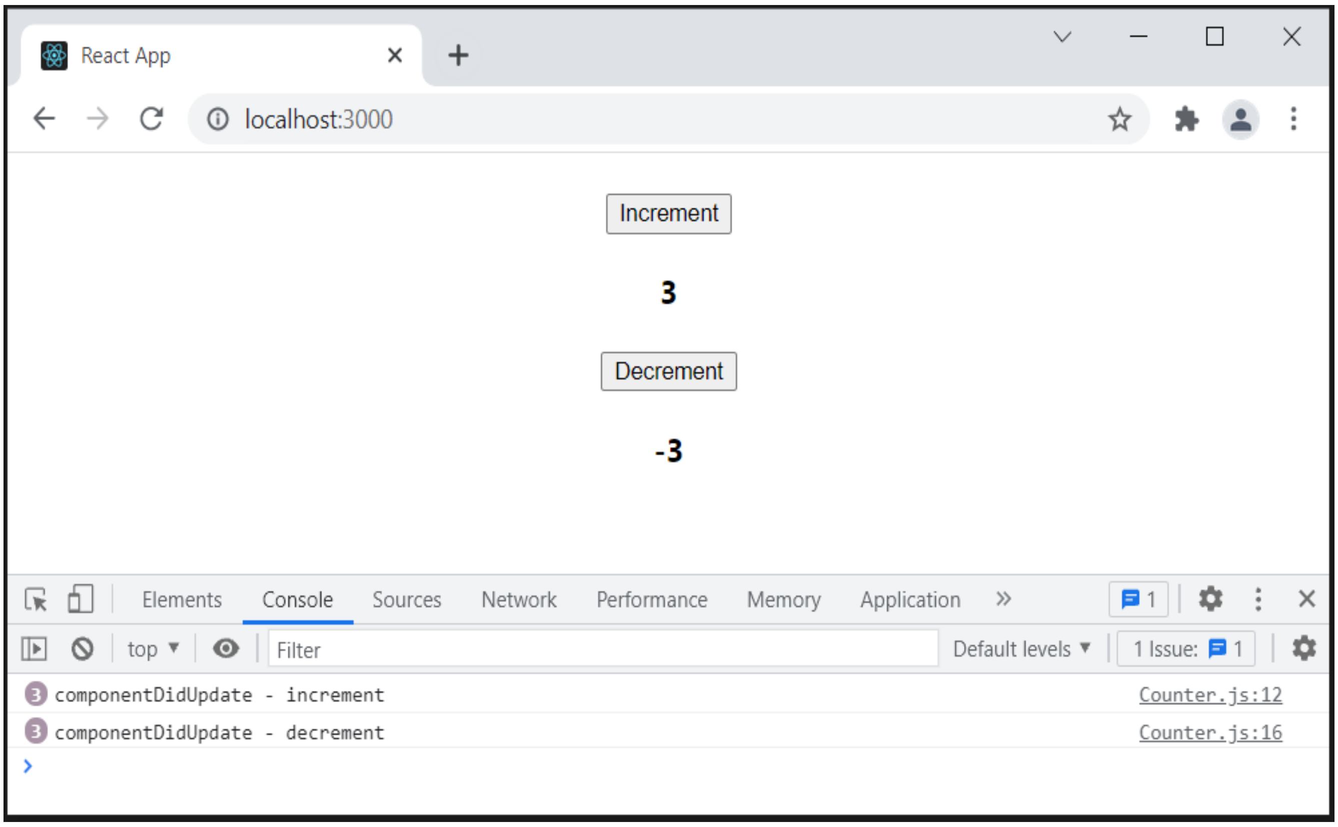
```
componentDidMount                    Counter.js:8
componentDidUpdate - increment       Counter.js:12
componentDidUpdate - decrement       Counter.js:16
```

When the “increment” button is clicked:



```
3 componentDidUpdate - increment                                    Counter.js:12
```

When the “decrement” button is clicked:



```
componentDidUpdate - increment          Counter.js:12
componentDidUpdate - decrement          Counter.js:16
```

This is how useEffect works with an empty array, an array with values, or no array.

Let’s understand how to perform componentWillUnmount with the useEffect hook.

```
import { useState, useEffect } from "react";

const Parent = () => {
  const [flag, setFlag] = useState(False);

  return (
    <>
      <button onClick={() => setFlag(!flag)}>Toggle child</button>
      {flag ? <Child /> : None}
    </>
  );
};
```

```
const Child = () => {
  useEffect(() => {
    return () => {
      console.log("Component unmounted");
    };
  }, []);

  return <h2>Child component</h2>;
};

export default Parent;
```

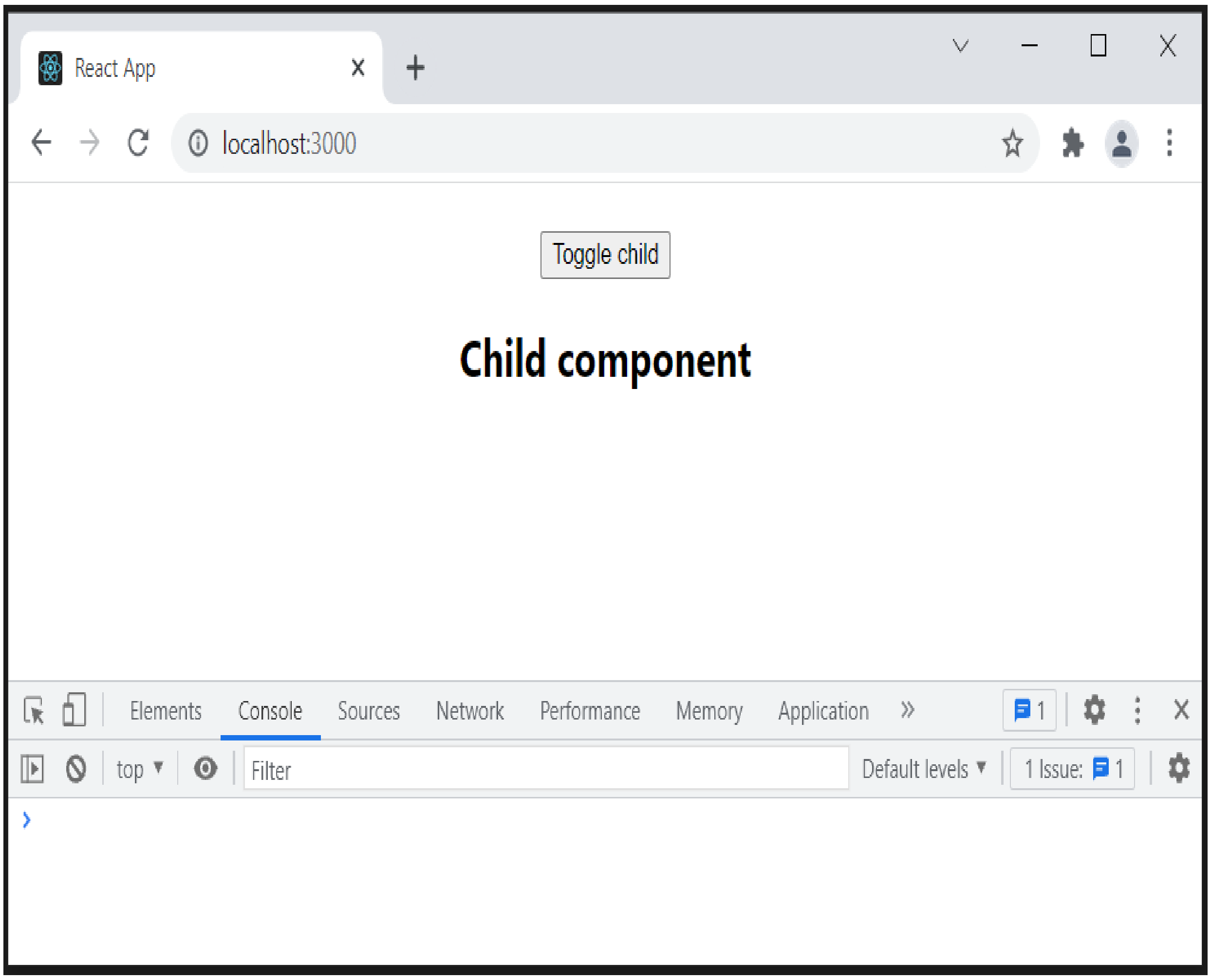
In the above code, the Parent component has a button that updates the value of flag. Initially, the value of flag is False, so the Child is not rendered. When the button is clicked, the value of flag changes to True, the Child component appears. On the next click, the value changes to False, and the Child component disappears, thus unmounting it.

In the Child component, the useEffect hook is used and it returns a function. This means the function will execute when the Child is unmounted.

```
useEffect(() => {
  return () => {
    console.log("Component unmounted");
  };
}, []);
```

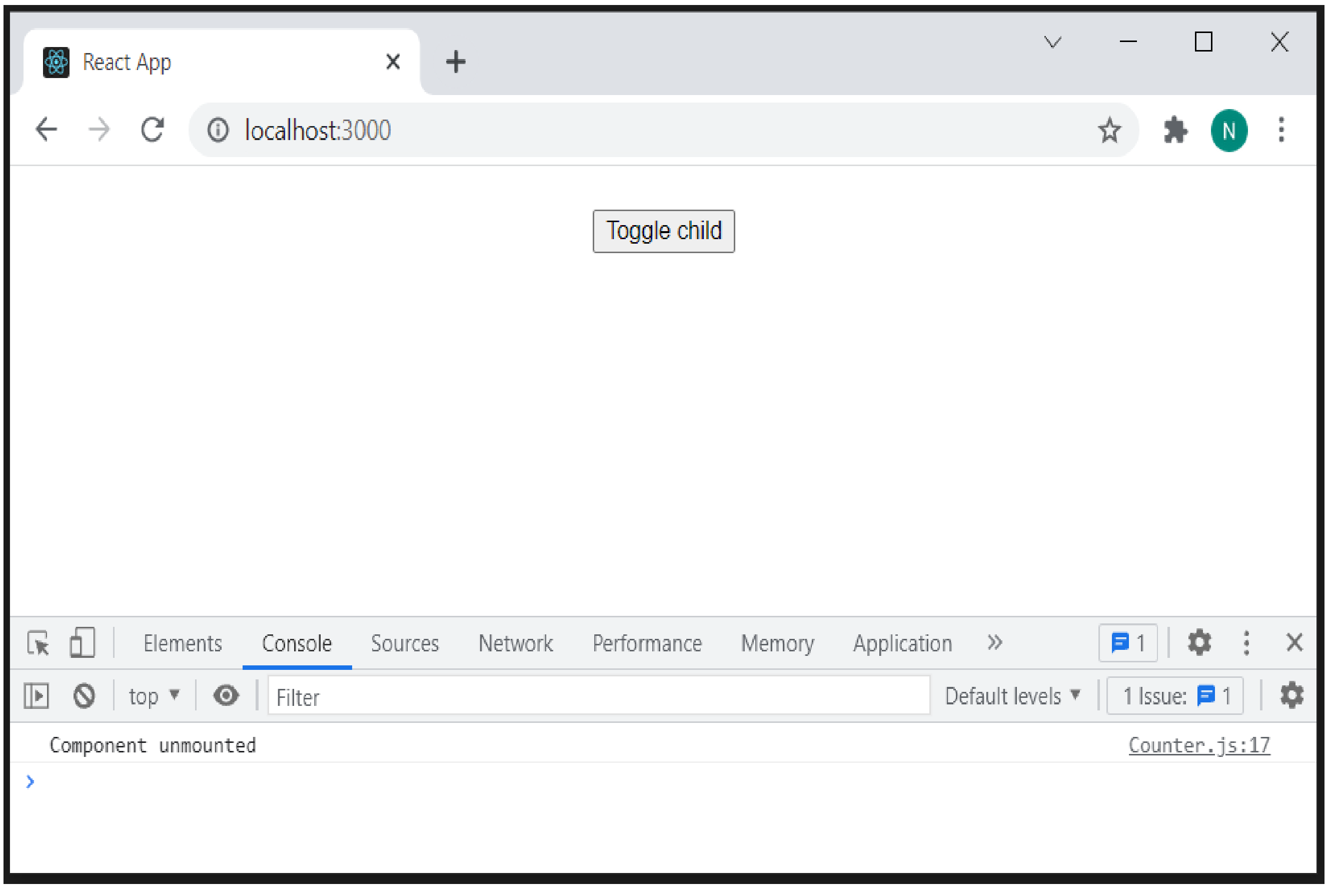
Let’s check the output.

When the button is clicked.



Child component

When the button is clicked again:



```
Component unmounted                                        Counter.js:17
```

When the component is unmounted, the statement written in the returning function appears in the console.

Wrapping it up

Functional components are easier to use, understand and read than the class components. The introduction of React hooks is not less than a revolution because it shifted the focus from class components to functional components.

The useState hook is probably the most popular and commonly used React hook. Apart from useState, useReducer can be used for state management. Other React hooks such as useEffect are also useful in several ways. There are several other react hooks that could be used in functional components for easier React development.

In this tutorial, we discussed what are functional components and how to use them with state, props, and lifecycle methods.