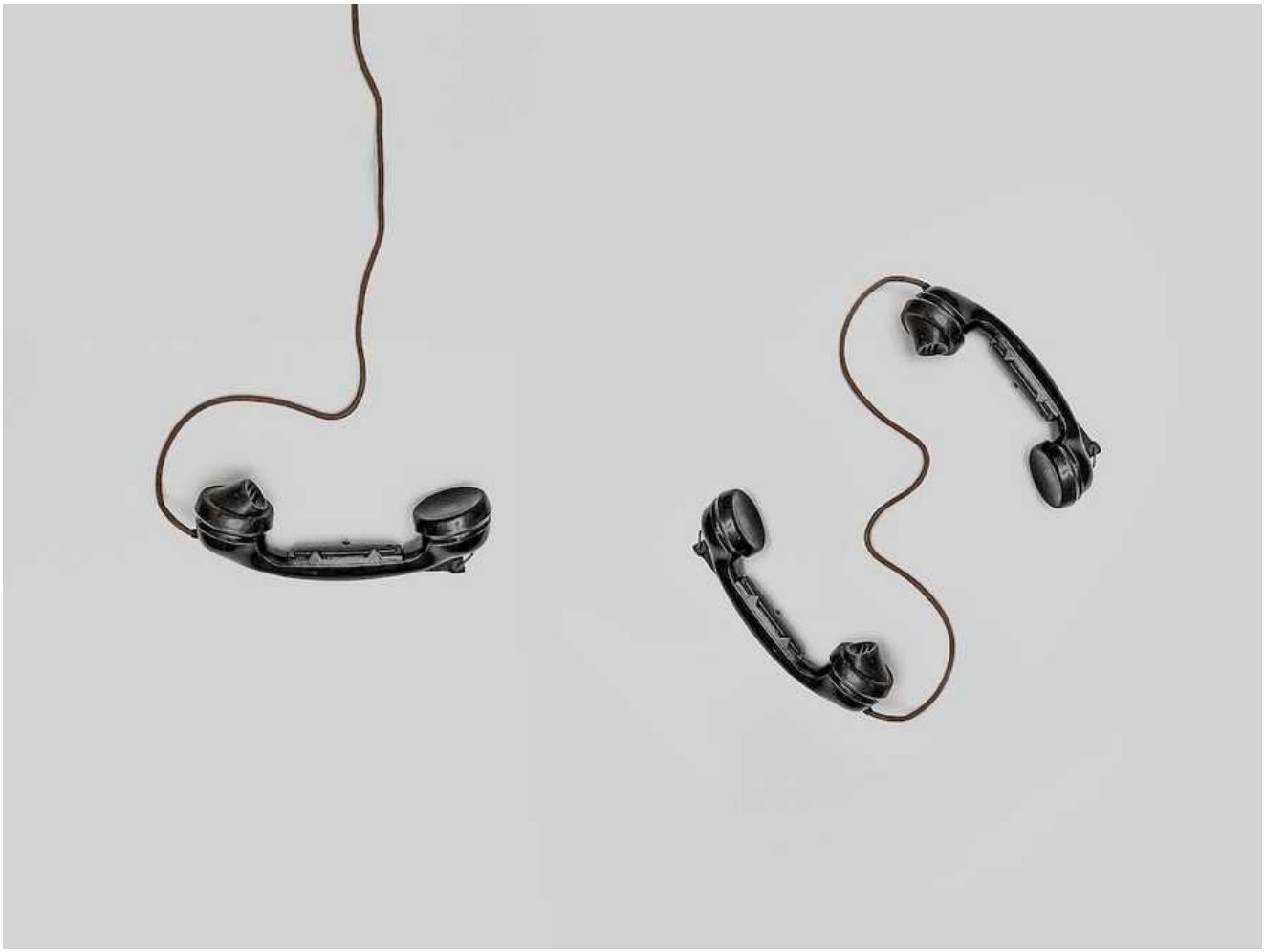
React Event Handlers: onClick, onChange ...

JANUARY 15, 2020 BY ROBIN WIERUCH - EDIT THIS POST



In this React tutorial, we will get to know event handlers in React for HTML elements such as button and input elements. You will learn how to use a button with its onClick event and how to define and use different kinds of event handlers. Essentially we will go through three kinds of event handlers: event handlers, inline event handlers and callback event handlers.

EVENT HANDLER IN REACT

First, we will start with a button example in React for a specific **onClick event handler**. It's the most basic example on how to handle events in React with an **event handler** (also called **event handler function** or **handler**). A button has a onClick attribute which receives a function. This function is called every time the event is triggered (here: when clicking the button):

```
import React from 'react';

function App() {
  function handleClick() {
    console.log('Button click ...');
  }

  return (
    <div>
      <button type="button" onClick={handleClick}>
        Event Handler
      </button>
    </div>
  );
}
```

It works similar for other attributes like onChange (onChange event handler) and onSubmit (onSubmit event handler). For beginners, often the onClick is not working, because instead of passing a function, they call the function directly in the JSX. For example, in the next version, the event handler is only called once when rendering the component for the first time. Every other click doesn't call the event handler function, because the function's return value is used in the onClick attribute and not the function itself. So there is nothing to call; unless the function returns another function:

```
import React from 'react';

function App() {
  function handleClick() {
    console.log('Button click ...');
  }

  // don't do this
  return (
    <div>
      <button type="button" onClick={handleClick()}>
        Event Handler
      </button>
    </div>
  );
}
```

By using a JavaScript arrow function, the event handler function can be made more concise. That's an optional step though. Personally, I like to have event handlers as arrow functions:

```
import React from 'react';

function App() {
  const handleClick = () => {
    console.log('Button click ...');
  };

  return (
    <div>
      <button type="button" onClick={handleClick}>
        Event Handler
      </button>
    </div>
  );
}
```

But once more event handlers add up in a React component, it's nice to make them more distinguishable from other variables by giving them the `function` statement again:

```
import React from 'react';

function App() {
  const user = {
    id: '123abc',
    username: 'Robin Wieruch',
  };

  function handleUserSignIn() {
    // do something
  }

  function handleUserSignUp() {
    // do something
  }

  function handleUserSignOut() {
    // do something
  }

  ...
}
```

After all, the event handler for the onClick event should implement some business logic. In this example, React's useState Hook is used to update some state via a onClick button event:

```
import React from 'react';

function App() {
  const [count, setCount] = React.useState(0);

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

  return (
    <div>
      Count: {count}

      <button type="button" onClick={handleClick}>
        Increase Count
      </button>
    </div>
  );
}
```

The next example shows you an input field instead of a button. There, we are using the actual **event** which is always passed as first parameter to the event handler function. The event is a **synthetic event** from React which essentially encapsulates the native HTML event and adds some functionality on top of it. This event gives you the value from the input field every time someone types into it with the event's target property:

```
import React from 'react';

function App() {
  const [text, setText] = React.useState('');

  function handleChange(event) {
    setText(event.target.value);
  }

  return (
    <div>
      <input type="text" onChange={handleChange} />

      {text}
    </div>
  );
}
```

Previously we haven't used the event, because in our button example we didn't need it. In the input field example we needed it though. Last but not least, don't forget to pass the value to the input element to make it a controlled component:

```
import React from 'react';

function App() {
  const [text, setText] = React.useState('');

  function handleChange(event) {
    setText(event.target.value);
  }

  return (
    <div>
      <input type="text" value={text} onChange={handleChange} />

      {text}
    </div>
  );
}
```

That's event handlers in a nutshell. Let's learn about more advanced handlers in React.

INLINE EVENT HANDLER IN REACT

Inline event handlers, also called **inline handlers**, give us lots of new options by using an event handler directly in JSX:

```
import React from 'react';

function App() {
  const [count, setCount] = React.useState(0);

  return (
    <div>
      Count: {count}

      <button
        type="button"
        onClick={function() {
          setCount(count + 1);
        }}
      >
        Increase Count
      </button>
    </div>
  );
}
```

Using the common function statement in JSX is verbose though. Hence, JavaScript arrow functions come in handy to define inline handlers more concise:

```
import React from 'react';

function App() {
  const [count, setCount] = React.useState(0);

  return (
    <div>
      Count: {count}

      <button
        type="button"
        onClick={() => setCount(count + 1)}
      >
        Increase Count
      </button>
    </div>
  );
}
```

In general, developers are lazy people, so often inline event handlers are used to avoid the extra function declaration outside the JSX. However, this moves lots of business logic into the JSX which makes it less readable, maintainable and error prone. Personally, I like to keep the JSX clean without inline event handlers and declare event handlers instead outside of the JSX.

Inline handlers are also used to pass a parameter to a more universal handler which is defined outside of the JSX:

```
import React from 'react';

function App() {
  const [count, setCount] = React.useState(0);

  function handleCount(delta) {
    setCount(count + delta);
  }

  return (
    <div>
      Count: {count}

      <button type="button" onClick={() => handleCount(1)}>
        Increase Count
      </button>
      <button type="button" onClick={() => handleCount(-1)}>
        Decrease Count
      </button>
```

```
    </div>
  );
}
```

This way, it's also possible to pass event and parameter side-by-side. Even though it's not needed in this example, you will surely experience one or the other case in the future where you will need the event (e.g. preventDefault for React Forms):

```
import React from 'react';

function App() {
  const [count, setCount] = React.useState(0);

  function handleCount(event, delta) {
    setCount(count + delta);
  }

  return (
    <div>
      Count: {count}

      <button type="button" onClick={event => handleCount(event, 1)}>
        Increase Count
      </button>
      <button type="button" onClick={event => handleCount(event, -1)}>
        Decrease Count
      </button>
    </div>
  );
}
```

So whenever you need to **pass event and parameter**, for instance when you need an extra parameter for your onClick event, inline event handlers may help you. Then a more universal event handler outside of the JSX can use this extra parameter(s).

CALLBACK EVENT HANDLER IN REACT

Last but not least, there are **callback event handlers** or **callback handlers** in short. They are used when a child component needs to communicate to a parent component. Since React props are only passed down the component tree, a callback handler, which is a function at its core, is used to communicate upward:

```
import React from 'react';
```

```
function App() {
  const [text, setText] = React.useState('');

  // 1
  function handleTextChange(event) {
    setText(event.target.value); // 3
  }

  return (
    <div>
      <MyInput inputValue={text} onInputChange={handleTextChange} />

      {text}
    </div>
  );
}

// 2
function MyInput({ inputValue, onInputChange }) {
  return (
    <input type="text" value={inputValue} onChange={onInputChange} />
  );
}
```

A callback handler is defined somewhere (1), used somewhere else (2), but calls back to the place where its defined (3). This way, it's possible to communicate from child to parent components. A callback handler is passed down via React props and communicates up when the function is called.

. . .

You have learned about React's event handler, inline event handler, and callback event handler and how to use them in buttons for their onClick events and in input fields for their onChange events. Also there are other event handlers, for instance the onSubmit for a form element where the event is actually needed to prevent the native browser behavior. Anyway, all of these event handlers have their specific purpose. Your goal should be to keep the your code readable and maintainable, so get you and your team on the same page for a code style in React here. You can find a playground for React event handlers on GitHub.

Show Comments

KEEP READING ABOUT REACT›

WHAT IS PREVENTDEFAULT() IN REACT?

React uses synthetic events to handle events from button, input and form elements. A synthetic event is a shell around the native DOM event with additional information for React. Sometimes you have to...

MEDIATOR COMPONENT IN REACT

When I work with clients on their React applications, I often encounter convoluted React components. Such overly complex components happen, because they have too many responsibilities -- whereas one...