Event Handling in React

(https://github.com/learn-co-curriculum/react-hooks-event-handling) (https://github.com/learn-co-curriculum/react-hooks-event-handling/issues/new)

Learning Goals

- Understand how to use React events in our application
- Attach event listeners to JSX elements

Overview

In this lesson, we'll cover the event system in React. We'll learn how to attach event listeners to specific elements.

Adding Event Handlers

Consider the following component:

```
function Tickler() {
  function tickle() {
    console.log("Teehee!");
  }

  return <button>Tickle me!</button>;
}
```

We have a `tickle()` function, but no way to trigger it! This is a perfect time to add an event listener so that we can see the message in our console. We attach an event listener to an element much like we add a prop.

The listener name is always comprised of `on`, and the event name itself, for example `click`. These are joined together and camel-cased, so if we wanted to add a click handler, we'd call the prop `onClick`.

This prop takes a **callback function** as a value. It can either be a reference to a function (like our `tickle()` function), or an inline function. Most of the time, we'll use a function reference.

All together, it looks like this:

```
<button onClick={tickle}>Tickle me!</button>
```

As you can see, we're passing a function *reference*, and not executing the `tickle` function. Our updated component looks like this:

```
function Tickler() {
  function tickle() {
    console.log("Teehee!");
  }

  return <button onClick={tickle}>Tickle me!</button>;
}
```

Now, when we click the button, we see a message in our console. Awesome!

We can also use an arrow function to define an event handler inline, and pass it to the event listener:

```
function Tickler() {
  return <button onClick={() => console.log("Teehee!")}>Tickle me!</button>;
}
```

Arrow functions are a good choice if your event handler doesn't need to handle much logic. If you have more than one line of code to run in your event handler, it's a good idea to create a separate callback function (like in the first example).

What Can We Listen For Events On?

One important thing to note about event listeners: you can only attach them to DOM elements, *not* React components. For example, this will not work:

```
function Clickable() {
  return <button>Click Me</button>;
}

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

  return <Clickable onClick={handleClick} />;
}
```

... but this will:

```
function Clickable() {
  function handleClick() {
    console.log("click");
  }
  return <button onClick={handleClick}>Click Me</button>;
}

function App() {
  return <Clickable />;
}
```

If we want to make the first example work so that `handleClick` is called in `App`, we'd have to pass `onClick` as a **prop** on the `Clickable` component, like so:

```
function Clickable({ onClick }) {
  return <button onClick={onClick}>Click Me</button>;
}
```

```
function App() {
  function handleClick() {
    console.log("click");
  }

  return <Clickable onClick={handleClick} />;
}
```

In this example, we're passing down a reference to the `handleClick` function as a **prop** called `onClick` to the `Clickable` component. Then, we're using that prop as the callback function for the `<button>` element's `onClick` attribute. That way, when the `<button>` element is clicked, the callback function `handleClick` will be called.

Whew! That's a lot to keep track of. We'll cover this concept of passing down callback functions as props in more detail later on.

Let's explore a few other common event types and their use cases here. There's some starter code provided, so feel free to code along and test things out in the console!

onClick

As we saw in the example above, adding a `click` event is pretty straightforward!

Using our `Tickler` component as an example, let's see what else we can do with a click event.

Update your component to look like this:

```
function Tickler() {
  function tickle(event) {
    console.log(event);
  }

  return <button onClick={tickle}>Tickle me!</button>;
}
```

Just as in JavaScript, when we handle events in React, we can provide an `event` parameter to our event handler callback function. When the button is clicked, we can access all the information about the event (the event `target`, mouse coordinates via `clientX` and `clientY`, etc).

What if we wanted to pass other information to the event handler though? In the `MultiButton` component, we have three buttons that all share the same callback function for their `onClick` event:

```
function MultiButton() {
  function handleClick(number) {
    console.log(`Button ${number} was clicked`);
  }

  return (
    <div>
      <button onClick={handleClick}>Button 1</button>
      <button onClick={handleClick}>Button 2</button>
      <button onClick={handleClick}>Button 3</button>
    </div>
  );
}
```

When one of the buttons is clicked, we want the callback to log out the button's number. If you try clicking one of those buttons now, you'll still see the `event` object being logged, not the number of the button.

We could try this:

```
function MultiButton() {
  function handleClick(number) {
    console.log(`Button ${number} was clicked`);
  }

  return (
    <div>
      <button onClick={handleClick(1)}>Button 1</button>
```

```
        <button onClick={handleClick(2)}>Button 2</button>
        <button onClick={handleClick(3)}>Button 3</button>
      </div>
    );
  }
```

...but now, the console messages will appear as soon as our component is rendered, not when the button is clicked. This is why we always need to provide a *function definition*, not a *function invocation* to our event handlers. Here's the solution:

```
function MultiButton() {
  function handleClick(number) {
    console.log(`Button ${number} was clicked`);
  }

  return (
    <div>
      <button onClick={() => handleClick(1)}>Button 1</button>
      <button onClick={() => handleClick(2)}>Button 2</button>
      <button onClick={() => handleClick(3)}>Button 3</button>
    </div>
  );
}
```

By writing out an arrow function here, we're providing each of our button's `onClick` handlers a *function definition* that will only be *invoked* when the button is clicked.

onChange

The `onChange` attribute is useful for handling changes to *input values*. This event listener is often used with `<input>`, `<select>`, and `<textarea>` inputs (basically, anywhere you need to capture a user's input).

Here's an example of using the `onChange` handler:

```
function ChangeItUp() {
  function handleChange(event) {
    console.log(`${event.target.name}: ${event.target.value}`);
  }

  return (
    <div>
      <input
        type="text"
        name="search"
        onChange={handleChange}
        placeholder="Enter search term..."
      />

      <select name="filter" onChange={handleChange}>
        <option value="all">Select a filter...</option>
        <option value="completed">Completed</option>
        <option value="incomplete">Incomplete</option>
      </select>
    </div>
  );
}
```

Try it out and note that, as text is entered into the `<input>` field, its value is captured using `event.target.value` and logged to the console.

onSubmit

Whenever you're working with `<form>` elements, handling the submit event is a good way to interact with all the data from the form after it's been submitted.

Here's a quick example:

```
function Login() {
  function handleSubmit(event) {
    event.preventDefault();
    console.log("I submit");
  }

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" name="username" placeholder="Enter username..." />
      <input type="password" name="password" placeholder="Enter password..." />
      <button>Login</button>
    </form>
  );
}
```

When the form is submitted, just like in vanilla JavaScript, you must call `event.preventDefault()` to prevent the form from making a network request.

We'll go into forms in more detail in a later lesson, and show the preferred way for collecting data from *all* the form input fields. For now, just remember: use the `onSubmit` event handler, and always call `preventDefault()` !

How Does React Handle Events?

You may have noticed when inspecting the `event` object that it's a bit different than the standard browser event. React's `event` object is a special object called: `SyntheticBaseEvent` .

React has its own event system with special event handlers called `SyntheticEvent` . The reason for having a specific event system instead of using native events is cross-browser compatibility. Some browsers treat events differently, and by wrapping these events into a consistent API, React makes our lives a lot easier. It's important to keep in mind that they are the *exact same events*, just implemented in a consistent way! That means these events also have methods that you can call like `preventDefault()` , `stopPropagation()` , and so on.

Conclusion

In React, you can add event listeners to elements in JSX by providing an `onEvent` attribute and passing a *callback function* to be used as an event handler. Some commonly used event listeners include `onClick`, `onChange`, and `onSubmit`. You can find a full list of supported events **here (https://reactjs.org/docs/events.html#supported-events)**.

React has its own internal events system that makes events behave consistently across various browsers.

Resources

- **Handling Events (https://reactjs.org/docs/handling-events.html)**
- **React Synthetic Events (https://reactjs.org/docs/events.html)**
- **Supported Events (https://reactjs.org/docs/events.html#supported-events)**