Events

How to respond to events in React

tl;dr

- JSX strips out the native HTML events and replaces them with their own.
- They're called synthetic events and they don't behave exactly like their native mirrors
- You create events on your own components by running a prop in the inner that was defined on the host.

React has created its own version of most events

- This is a place where React is very opinionated. :-(
- Normalized - to eliminate browser differences :-)

Some events that React can handle

- blur
- change
- click
- copy
- cut
- dbl-click
- focus
- keydown
- keypress
- keyup
- mousedown
- mouseenter
- mouseleave
- mousemove
- mouseout
- mouseover
- mouseup
- paste
- submit
- ...

... basically every event that is in a component's scope, React has an interface to.

Uppercase the first letter and precede it by *on*

```
<any onFoo={bar}></any>
```

Hey, React! When the user triggers the *foo* event, run the *bar* function.

Examples:

```
<button onClick={doIt}>
  Press me</button>
<img onMouseOver={count} />
<input onBlur={go}
       onKeyUp={run} />
```

But the native browser events are stripped out by React

- So if you write

```
<MyComponent onclick="alert('foo')" />
```

- Your onclick will be ignored.
- We are forced to use React's synthetic events

Mouse events

- onClick
- onDoubleClick
- onMouseDown
- onMouseEnter
- onMouseLeave
- onMouseMove
- onMouseOver
- onMouseUp

```
<button onClick={processOrder}>
Go</button>
<img src="..."
 onDoubleClick={viewProduct} />
<img src="..."
  onMouseOver={incrementCounter} />
```

Form events

- onFocus
- onBlur
- onChange
- onCut
- onCopy
- onPaste
- onSubmit
- onKeyDown
- onKeyUp
- onKeyPress

```
<input onFocus={checkAllFields}
   onBlur={checkAgain}
   onKeyUp={getSuggestions} />
```

- React adds these synthetic events to W3C elements (aka NOT your components!)
- Thus you can't have a, say, click event on your component. Just on the HTML elements inside your component.*

* Unless you create your own custom event. More on that later.

Even when Synthetic events appear to match their native counterparts, they're still different

- Examples: onchange fires only when the user commits a value (via blur for example) but onChange event fires on every keystroke.
- There's a native ondblclick but not a React onDblClick event. It is onDoubleClick. (sheesh!)
- And there are other peculiarities ...

There are unsupported events

- They usually fall in three categories

1. Window- and Browser-level events
   - beforePrint, hashChange, resize, message, DOMContentLoaded, beforeunload, load,
2. Experimental events
   - They eventually get support after they're mainstream
   - (eg. Device events, Touch events, pointer events are new-ish)
3. Events that just don't make sense to do
   - reset (for forms), wheel

The event object is reused!

- When an event fires, an event object is created. Then React creates a Synthetic event object. This is expensive.
- So to increase performance, the Synthetic event object is reused over and over.
- This would not be a good idea:

```
function handleClick(event) {
  let name = event.target.name;
  setTimeout(function () {
    console.log(name);
  }, 1000);
}
```

- Because one second later, the event.target object would point to a completely different object.
- event.persist() would cause it to save the value.

Passing values to the handler

Say you have an event handler function:

MyComponent.js

```
function addPerson(person) {
  console.log("Person was added", person);
  try {
   insertIntoDB(person);
   return true;
  } catch {
   return false;
  }
}
```

And you have some JSX:

MyComponent.js

```
let person = {};
return (
 <form onSubmit={addPerson}>
  <input value={person.first} />
  <input value={person.last} />
  <input type="submit" />
 </form>
);
```

How does the person object get sent to addPerson?!?

- When you specify an event handler, you're not running a function.
- You're registering a function to be run later.
- You must pass a function to the event

```
console.log(typeof addPerson);   // function
console.log(typeof addPerson(person));   // bool
```

So to pass a parameter, use an arrow function

MyComponent.js

```
let person = {};
return (
 <form onSubmit={() => addPerson(person)}>
  <input value={person.first} />
  <input value={person.last} />
  <input type="submit" />
 </form>
);
```

And to pass the event object ...

MyComponent.js

```
...
return (
 <button onDoubleClick={e => doStuff(e, obj1, obj2)}>
  Click me!
 </button>
);
```

- This works because when an event is triggered, the (synthetic) event object is passed into the registered function

How to create your own custom events

In the inner component...

InnerComponent.js

```
return <div>
 {/*
  Bunch of JSX here. The user interacts and some
  condition arises and we call raiseEvent()
 */}
</div>
function raiseEvent() {
  props.onCustomEvent();
}
```

Then in the host you can do this...

HostComponent.js

```
return <div>
 <InnerComponent onCustomEvent={doStuff} />
</div>
function doStuff() {
  // This is where you'd process the custom
event.
}
```

tl;dr

- JSX strips out the native HTML events and replaces them with their own.
- They're called synthetic events and they don't behave exactly like their native mirrors
- You create events on your own components by running a prop in the inner that was defined on the host.