Day 7

Handling Events In Reactjs

All About Events In React

@oluwakemi Oluwadahunsi

What Are Events?

Events are **actions** or **occurrences** that happen in the browser, like clicking a button, submitting a form, pressing a key, or scrolling a page.

In React, we handle these events to make our applications dynamic and interactive. React wraps native browser events to provide a cross-browser interface called the **SyntheticEvent**.

Handling events in React is quite straightforward.

Here's the basic syntax:

- React events are named using **camelCase** (e.g., onClick, onChange).
- Event handlers are passed as **functions** (not strings, unlike in plain HTML).

@oluwakemi Oluwadahunsi

For Example:

```jsx
import React from 'react';

const App = () => {
  const handleClick = () => {
    alert('Button was clicked!');
  };

  return (
    <div>
      <h1>Handling Events in React</h1>
      {/* Attach the event handler to the button */}
      <button onClick={handleClick}>Click Me!</button>
    </div>
  );
};

export default App;
```

In this example:

- The **handleClick** function is the event handler, and it is called when the button is clicked.

- The event handler is attached to the button using the **onClick** attribute.

@oluwakemi Oluwadahunsi

Event Handling with Parameters

If you need to pass parameters to an event handler, you can use an arrow function or the bind method.

```jsx
import React from 'react';

const App = () => {
  const handleClick = (message) => {
    alert(message);
  };

  return (
    <div>
      <button onClick={() => handleClick('Hello, React!')}>Show Message</button>
    </div>
  );
};

export default App;
```

- The arrow function **(message) => alert(message)** is used to pass a parameter to the handleClick function.
- When the button is clicked, it displays an alert with the message "Hello, React!".

@oluwakemi Oluwadahunsi

Event Object in React

React provides a special object called **SyntheticEvent** that normalizes events across different browsers. It behaves like the native event object in JavaScript but works consistently across all browsers.

```jsx
import React from 'react';

const App = () => {
  const handleChange = (event) => {
    console.log('Event Type:', event.type);
    console.log('Event Target Value:', event.target.value);
  };

  return (
    <div>
      <input type="text" onChange={handleChange} placeholder="Type something..." />
    </div>
  );
};

export default App;
```

- The **handleChange** function receives the event object automatically when the input field changes.
- You can access various properties of the event object, like event.type (the type of event) and event.target.value (the current value of the input field).

@oluwakemi Oluwadahunsi

Binding Event Handlers in Class Components

This refers to the process of ensuring that the event handler methods (functions) have the correct **this** context when they are called.

In JavaScript, the value of **this** inside a function depends on how the function is called.

In a class component, if you directly assign an event handler method (like **this.handleClick**) to an event (like **onClick**), the value of **this** inside **handleClick** will be undefined or refer to the wrong object when the method is invoked.

This is because this in JavaScript is **not automatically** bound to class instances.

To fix this, we **bind** the event handler method to the component instance (**this**) in the constructor.

@oluwakemi Oluwadahunsi

For Example: Binding in the constructor

```jsx
import React, { Component } from 'react';

class ClickButton extends Component {
  constructor(props) {
    super(props);
    // Bind the event handler to the current instance of the component
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    console.log('Button clicked!', this);
    // 'this' will refer to the current component instance
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        Click Me
      </button>
    );
  }
}

export default ClickButton;
```

The **handleClick** method is bound to the component instance using **.bind(this)** inside the constructor. This ensures that whenever **handleClick** is called, this will refer to the correct instance of the component.

@oluwakemi Oluwadahunsi

An alternative method is to define your event handler as an **arrow function**, which automatically binds **this** to the class instance like this:

```
handleClick = () ⇒ {
  console.log('Button clicked!', this);
};
```

This eliminates the need for binding in the constructor.

Commonly Used Events in React

Here are some of the most commonly used events in React:

- **Mouse Events:** onClick, onDoubleClick, onMouseEnter, onMouseLeave, onMouseMove, onMouseDown, onMouseUp, onMouseMove, onMouseOver, onMouseOut.
- **Keyboard Events:** onKeyDown, onKeyUp, onKeyPress.
- **Form Events:** onChange, onSubmit, onFocus, onBlur, onReset.
- **Clipboard Events:** onCopy, onCut, onPaste.
- **Composition Events:** onCompositionStart, onCompositionUpdate, onCompositionEnd.
- **Clipboard Events:** onCopy, onCut, onPaste.
- **Input Events:** onInput, onInvalid.
- **Touch Events:** onTouchCancel, onTouchStart, onTouchMove, onTouchEnd

@oluwakemi Oluwadahunsi

Preventing Default Behavior

To prevent the default behavior of an event in React, you use the **preventDefault()** method on the event object.

Example: Preventing Default Form Submission using **event.preventDefault()** in the form's onSubmit event handler.

```jsx
import React from 'react';

const App = () => {
  const handleSubmit = (event) => {
    event.preventDefault(); // Prevents form submission
    alert('Form submitted!');
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" placeholder="Enter your name" />
      <button type="submit">Submit</button>
    </form>
  );
};

export default App;
```

event.preventDefault() is used to prevent the default behavior of the form submission, which is to reload the page.

@oluwakemi Oluwadahunsi

Using Synthetic Events: Handling Multiple Events

You can use **Synthetic Events** to handle multiple events in a single handler, making the code more reusable and efficient.

```jsx
import React from 'react';

const App = () => {
  const handleEvent = (event) => {
    if (event.type === 'mouseenter') {
      console.log('Mouse entered!');
    } else if (event.type === 'mouseleave') {
      console.log('Mouse left!');
    }
  };

  return (
    <div onMouseEnter={handleEvent} onMouseLeave={handleEvent}>
      Hover over me!
    </div>
  );
};

export default App;
```

- The **handleEvent** function handles both **mouseenter** and **mouseleave** events using a single function.
- The event type is determined by **event.type**.

@oluwakemi Oluwadahunsi

Using Inline Event Handlers vs. Separate Functions

React allows you to define event handlers inline or separately as functions.

Inline handlers can be useful for simple actions, while separate functions make your code cleaner and more readable for complex logic, but best is to use as **functions** when handling complex events.

Example of Inline Event Handler

```
1 import React from 'react';
2
3 const App = () ⇒ {
4   return (
5     <button onClick={() ⇒ alert('Button clicked!')}>
6       Click Me!
7     </button>
8   );
9 };
10
11 export default App;
```

inline handler

@oluwakemi Oluwadahunsi

Example of Separate Event Handler Function

```jsx
import React from 'react';

const App = () => {
  const showAlert = () => {
    alert('Button clicked!');
  };

  return (
    <button onClick={showAlert}>
      Click Me!
    </button>
  );
};

export default App;
```

Separate Function

function called

@oluwakemi Oluwadahunsi

Event Delegation in React

React uses a form of event delegation by attaching event listeners at the root of the DOM tree.

This helps in improving performance by **minimizing** the number of event handlers and leveraging event bubbling.

React automatically handles this behind the scenes, so you don’t need to worry about manually attaching event listeners.

@oluwakemi Oluwadahunsi

Handling Events with Hooks in Functional Components

With functional components, Hooks like **useState** and **useEffect** provide a powerful way to handle events and manage state and side effects.

For Example:

```jsx
import React, { useState } from 'react';

const App = () => {
  const [count, setCount] = useState(0);

  const incrementCount = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={incrementCount}>Increment</button>
    </div>
  );
};

export default App;
```

- **useState** is used to manage the state **(count)**.
- The **incrementCount** function is called on button click to update the state.

@oluwakemi Oluwadahunsi

Best Practices for Handling Events in React

1. Use Descriptive Event Handler Names: Name your event handlers descriptively to make your code more readable (e.g., handleClick, handleSubmit).

2. Use Separate Functions for Complex Logic: Keep your components clean by moving complex event logic to separate functions.

3. Minimize Inline Event Handlers: Prefer using separate functions over inline handlers to avoid unnecessary re-renders.

4. Leverage the Event Object: Make full use of the **SyntheticEvent** object to handle events efficiently and access useful properties.

5. Utilize Hooks in Functional Components: Use hooks like **useState** and **useEffect** to manage state and side effects in functional components.

@oluwakemi Oluwadahunsi

I hope you found this material useful and helpful.

Remember to:

Like

Save for future reference

&

Share with your network, be helpful to someone 👌

@oluwakemi Oluwadahunsi

Hi There!

Thank you for reading through

Did you enjoy this knowledge?

💼 Follow my LinkedIn page for more work-life balancing and Coding tips.

🌐 LinkedIn: Oluwakemi Oluwadahunsi

kodemaven-portfolio.vercel.app