React Event Handling

Event handling is a fundamental aspect of building interactive user interfaces in React. Events such as clicks, key presses, mouse movements, etc., trigger actions within React components. Properly handling these events is crucial for creating responsive and interactive applications.

React components can listen and respond to various events similar to HTML event handling.

Events are handled using event handlers, which are functions that are executed when a specific event occurs.

Here are some examples in React demonstrating how to listen and respond to various events.

1. Handling Click Events:

```
import React from 'react';

class ClickExample extends React.Component {
  handleClick() {
    alert('Button clicked!');
  }

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

export default ClickExample;
```

In this example, when the button is clicked, the handleClick method is invoked, which displays an alert.

2. Handling Input Events (onChange):

```
import React from 'react';

class InputExample extends React.Component {
  constructor(props) {
    super(props);
    this.state = { value: '' };
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(event) {
    this.setState({ value: event.target.value });
  }

  render() {
```

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

export default InputExample;
```

In this example, the handleChange method is called whenever the input field's value changes. It updates the component's state with the new value entered by the user.

3. Handling Form Submission:

```
import React from 'react';

class FormExample extends React.Component {
  constructor(props) {
    super(props);
    this.state = { username: '', password: '' };
    this.handleSubmit = this.handleSubmit.bind(this);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(event) {
    this.setState({ [event.target.name]: event.target.value });
  }

  handleSubmit(event) {
    event.preventDefault();
    console.log('Submitted:           ',           this.state.username,
this.state.password);
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <input
          type="text"
          name="username"
          value={this.state.username}
          onChange={this.handleChange}
          placeholder="Username"
        />
        <input
          type="password"
```

```
          name="password"
          value={this.state.password}
          onChange={this.handleChange}
          placeholder="Password"
        />
        <button type="submit">Submit</button>
      </form>
    );
  }
}

export default FormExample;
```

In this example, the handleSubmit method is called when the form is submitted. It prevents the default form submission behavior and logs the username and password to the console.

4. Handling Mouse Events:

```
import React from 'react';

class MouseExample extends React.Component {
  handleMouseEnter() {
    console.log('Mouse entered');
  }

  handleMouseLeave() {
    console.log('Mouse left');
  }

  render() {
    return (
      <div
        onMouseEnter={this.handleMouseEnter}
        onMouseLeave={this.handleMouseLeave}
        style={{ width: '200px', height: '200px', backgroundColor:
'lightgray' }}
      >
        Hover over me
      </div>
    );
  }
}

export default MouseExample;
```

In this example, the handleMouseEnter method is called when the mouse enters the element, and the handleMouseLeave method is called when the mouse leaves the element.

5. Handling Keyboard Events:

```
import React from 'react';

class KeyboardExample extends React.Component {
  handleKeyPress(event) {
    console.log('Key pressed: ', event.key);
  }

  render() {
    return (
      <input
        type="text"
        onKeyPress={this.handleKeyPress}
        placeholder="Type something..."
      />
    );
  }
}

export default KeyboardExample;
```

6. Handling Focus Events:

```
import React from 'react';

class FocusExample extends React.Component {
  handleFocus() {
    console.log('Input field focused');
  }

  handleBlur() {
    console.log('Input field blurred');
  }

  render() {
    return (
      <input
        onFocus={this.handleFocus}
        onBlur={this.handleBlur}
        placeholder="Click here to focus..."
      />
    );
  }
}

export default FocusExample;
```

In this example, the handleFocus method is called when the input field receives focus, and the handleBlur method is called when the input field loses focus.

Synthetic Events

In React, Synthetic Events are a cross-browser wrapper around the browser's native event system. They are provided by React to ensure consistent behavior across different browsers and platforms.

Usage:

Accessing Event Properties:

- Synthetic Events provide the same properties and methods as native browser events.
- Developers can access properties like event.target, event.type, event.preventDefault(), etc., just like they would with native events.

Passing Events to Handlers:

- Synthetic Events are passed as arguments to event handler functions in React components.
- Developers can access the event object by specifying it as a parameter in the event handler function.

Example:

```
function handleClick(event) {
  console.log('Clicked at:', event.clientX, event.clientY);
}

<button onClick={handleClick}>Click me</button>
```

In this example, the handleClick function receives a Synthetic Event object as its argument when the button is clicked. It can access properties like clientX and clientY to get the coordinates of the mouse click.

React Binding Event Handlers

In JavaScript, the value of this is determined dynamically based on how a function is called. This behavior can sometimes lead to unexpected results, especially when dealing with event handlers in React class components.

When you pass a method as an event handler in React, such as onClick={this.handleClick}, the method loses its original context, and this may not refer to the component instance as expected. This can cause errors or unexpected behavior when accessing this.state or this.props within the method.

To ensure that this refers to the component instance within event handler methods, you need to bind them explicitly in React class components.

In React class components, you need to bind event handler methods in the constructor or use arrow functions to automatically bind this.

Arrow functions automatically bind this to the component's context, making it a common choice for event handlers.

1. Binding in Class Components:

In React class components, you can bind event handler methods in the constructor using the bind method. This explicitly sets the value of this within the method to the component instance.

Example:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 0
    };

    // Binding the handleClick method
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    // Accessing state
    const currentValue = this.state.value;
    this.setState({ value: currentValue + 1 });
  }
  render() {
    return (
      <button onClick={this.handleClick}>Click me</button>
    );
  }
}
```

2. Using Arrow Functions:

Alternatively, you can use arrow functions to define event handlers. Arrow functions automatically bind this to the component's context where they are defined.

Example:

```
class MyComponent extends React.Component {
    constructor(props) {
    super(props);
    this.state = {
      value: 0
    };
  }
  handleClick = () => {
    const currentValue = this.state.value;
    this.setState({ value: currentValue + 1 });
  };
  render() {
    return (
      <button onClick={this.handleClick}>Click me</button>
    );
  }
}
```

Methods as Props

In React, methods can be passed as props from parent components to child components. This pattern, known as "Methods as Props," allows child components to communicate with their parent components by invoking the passed methods.

1. Communication Between Components:

- Methods as Props enable communication between parent and child components in React.
- They allow child components to trigger actions or pass data to their parent components.

2. Passing Methods as Props:

- Parent components can pass their methods as props to child components.
- These methods are then accessible within the child components, allowing them to be invoked when certain events occur.

3. Triggering Actions in Parent Components:

- Child components can call the passed methods to trigger actions in their parent components.
- This allows for dynamic behavior where child components can influence the state or behavior of their parent components.

Benefits:

1. Separation of Concerns:

- Methods as Props promote a separation of concerns by allowing components to encapsulate their own logic.
- Parent components remain focused on managing state and rendering UI, while child components handle specific actions or behaviors.

2. Reusability and Modularity:

- By passing methods as props, components become more reusable and modular.
- Child components can be easily reused in different parts of the application without tightly coupling them to specific parent components.

Example :

```
// ParentComponent.js
import React from 'react';
import ChildComponent from './ChildComponent';
```

```
class ParentComponent extends React.Component {
  handleChildClick() {
    console.log('Child clicked!');
  }

  render() {
    return (
      <ChildComponent onClick={this.handleChildClick} />
    );
  }
}

export default ParentComponent;
```

```
// ChildComponent.js
import React from 'react';

const ChildComponent = ({ onClick }) => (
  <button onClick={onClick}>Click me</button>
);

export default ChildComponent;
```

In this example:

The ParentComponent passes its handleChildClick method as a prop called onClick to the ChildComponent.

When the button in the ChildComponent is clicked, it invokes the onClick method, which triggers the handleChildClick method in the ParentComponent.

Conditional Rendering in React

Conditional rendering in React involves displaying different components or elements based on certain conditions. This flexibility allows for the creation of more interactive and personalized user interfaces.

Conditional rendering is typically controlled by the state or props of a component. When the state or props change, React re-renders the component, updating the UI based on the new conditions.

Conditional rendering is achieved using JavaScript expressions within JSX. Developers can use three common patterns like if statements, ternary operators, or logical && operators to conditionally render components or elements.

1. Using if-else statement

```
import React from 'react';

const UserGreeting = ({ isLoggedIn }) => {
  let greeting;
  if (isLoggedIn) {
    greeting = <h1>Welcome back!</h1>;
  } else {
    greeting = <h1>Please sign up!</h1>;
  }
  return greeting;
};

export default UserGreeting;
```

In this example, We declare a variable greeting and use an if-else statement to determine its value based on the value of the isLoggedIn prop.

If isLoggedIn is true, greeting is assigned the welcome message JSX element; otherwise, it is assigned the sign-up prompt JSX element.

Finally, we return the greeting variable, which contains the appropriate JSX element based on the condition.

2. Using ternary operator

```
import React from 'react';

const UserGreeting = ({ isLoggedIn }) => {
  return isLoggedIn ? (
    <h1>Welcome back!</h1>
  ) : (
    <h1>Please sign up!</h1>
```

```
  );
};

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isLoggedIn: false
    };
    this.handleLogin = this.handleLogin.bind(this);
    this.handleLogout = this.handleLogout.bind(this);
  }

  handleLogin() {
    this.setState({ isLoggedIn: true });
  }

  handleLogout() {
    this.setState({ isLoggedIn: false });
  }

  render() {
    const { isLoggedIn } = this.state;

    return (
      <div>
        <UserGreeting isLoggedIn={isLoggedIn} />
        <button onClick={this.handleLogin}>Login</button>
        <button onClick={this.handleLogout}>Logout</button>
      </div>
    );
  }
}

export default App;
```

In this example the App component manages the isLoggedIn state.

The UserGreeting component receives the isLoggedIn prop and conditionally renders a welcome message or a sign-up prompt based on its value.

Buttons are provided in the App component to simulate logging in and logging out, which update the isLoggedIn state accordingly.

3. Using logical && operator

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

```
const UserGreeting = ({ isLoggedIn }) => {
  return isLoggedIn && <h1>Welcome back!</h1>;
};

const App = () => {
  const [isLoggedIn, setIsLoggedIn] = React.useState(false);

  const handleLogin = () => {
    setIsLoggedIn(true);
  };

  const handleLogout = () => {
    setIsLoggedIn(false);
  };

  return (
    <div>
      <UserGreeting isLoggedIn={isLoggedIn} />
      {!isLoggedIn && <button onClick={handleLogin}>Login</button>}
      {isLoggedIn && <button onClick={handleLogout}>Logout</button>}
    </div>
  );
};

export default App;
```

In this example the UserGreeting component receives the isLoggedIn prop and conditionally renders the welcome message using the logical && operator. If isLoggedIn is true, it renders the welcome message; otherwise, it renders null.

In the App component, buttons for login and logout are conditionally rendered based on the value of isLoggedIn. If isLoggedIn is false, the login button is shown, and if isLoggedIn is true, the logout button is shown.

Clicking on the login button sets isLoggedIn to true, and clicking on the logout button sets isLoggedIn to false, effectively toggling between the login and logout states.

Form Handling: Handling User Input, Handling Form Submission

In React form handling, there are two primary aspects to consider: handling user input and handling form submission. Let's explore each of these with examples:

Handling User Input

Handling user input involves capturing the data entered by the user into form fields and updating the component's state accordingly. Here's an example demonstrating how to handle user input in a simple form with a single input field:

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

function UserInputForm() {
  const [inputValue, setInputValue] = useState('');

  const handleInputChange = (event) => {
    setInputValue(event.target.value);
  };

  return (
    <form>
      <label>
        Name:
        <input              type="text"              value={inputValue}
onChange={handleInputChange} />
      </label>
      <p>Typed value: {inputValue}</p>
    </form>
  );
}

export default UserInputForm;
```

In this example, We use the useState hook to create a state variable inputValue to store the value of the input field.

The handleInputChange function is called whenever the value of the input field changes. It updates the inputValue state with the new value entered by the user.

The input field's value is set to inputValue, ensuring that React controls the input field and its value.

Handling Form Submission

Handling form submission involves capturing the data entered by the user and performing some action, such as submitting the data to a server or processing it locally. Here's an example demonstrating how to handle form submission:

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

function FormWithSubmission() {
  const [formData, setFormData] = useState({
    name: '',
    email: '',
  });

  const handleInputChange = (event) => {
    const { name, value } = event.target;
    setFormData({
      ...formData,
      [name]: value,
    });
  };

  const handleSubmit = (event) => {
    event.preventDefault();
    // Here you can perform actions like submitting the form data to a
server
    console.log('Form submitted:', formData);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
        <input     type="text"     name="name"     value={formData.name}
onChange={handleInputChange} />
      </label>
      <label>
        Email:
        <input    type="email"    name="email"    value={formData.email}
onChange={handleInputChange} />
      </label>
      <button type="submit">Submit</button>
    </form>
  );
}

export default FormWithSubmission;
```

In this example, We use the useState hook to create a state variable formData to store the form data.

The handleInputChange function updates the corresponding field in the formData state whenever the value of any input field changes.

The handleSubmit function is called when the form is submitted. It prevents the default form submission behavior (which would cause a page reload) and logs the form data to the console. You can replace the console log with any action you want to perform with the form data.

Setting Up a JS Config for the Form, Creating a Custom Dynamic Input Component, Dynamically Create Inputs Based on JS Config

Setting Up a JS Config for the Form

Before creating forms, let's define a JavaScript configuration object for our form. This configuration will dictate what inputs are rendered dynamically.

```
const formConfig = [
  { id: 1, type: "text", name: "firstName", placeholder: "First Name"
},
  { id: 2, type: "text", name: "lastName", placeholder: "Last Name" },
  { id: 3, type: "email", name: "email", placeholder: "Email" },
];
```

Creating a Custom Dynamic Input Component

Now, let’s create a component that takes type, name, and placeholder as props and renders an input field accordingly.

```
function InputField({ type, name, placeholder }) {
  return <input type={type} name={name} placeholder={placeholder} />;
}
```

Dynamically Create Inputs Based on JS Config

Using the configuration we defined, we can map over the formConfig array to dynamically create input components.

```
function Form() {
  return (
    <form>
      {formConfig.map(field => (
        <InputField key={field.id} type={field.type} name={field.name}
placeholder={field.placeholder} />
      ))}
    </form>
  );
}
```

Handling User Input

To manage the state of the input fields, we use React’s useState to keep track of each input's value.

```
function Form() {
  const [formData, setFormData] = useState({ firstName: "", lastName:
"", email: "" });

  function handleChange(event) {
    const { name, value } = event.target;
    setFormData(prevState => ({
      ...prevState,
      [name]: value
    }));
  }

  return (
    <form>
      {formConfig.map(field => (
        <InputField
          key={field.id}
          type={field.type}
          name={field.name}
          placeholder={field.placeholder}
          value={formData[field.name]}
          onChange={handleChange}
        />
      ))}
    </form>
  );
}
```

Handling Form Submission

Finally, let's handle the form submission to log the form data.

```
function Form() {
  const [formData, setFormData] = useState({ firstName: "", lastName:
"", email: "" });

  function handleChange(event) {
    const { name, value } = event.target;
    setFormData(prevState => ({
      ...prevState,
      [name]: value
    }));
  }

  function handleSubmit(event) {
```

```
    event.preventDefault();  // Prevents the default form submit action
    console.log('Submitted Data:', formData);
  }

  return (
    <form onSubmit={handleSubmit}>
      {formConfig.map(field => (
        <InputField
          key={field.id}
          type={field.type}
          name={field.name}
          placeholder={field.placeholder}
          value={formData[field.name]}
          onChange={handleChange}
        />
      ))}
      <button type="submit">Submit</button>
    </form>
  );
}
```

Update the ‘contact us’ page to a ReactJS. An email should be sent each time contact us form is submitted.

App.js

```
import React from 'react';
import Lab7 from 'components/Lab7';

function App() {
  return (
      <Lab7/>
  );
}

export default UserInputForm;
```

Prerequisite:

```
npm install emailjs-com
```

Step 1: Sign Up / Log In

First, make sure you are signed up and logged into EmailJS:

Visit EmailJS and either sign up for a new account or log in if you already have one.

Step 2: Obtain Your User ID

Once logged in, your User ID can usually be found in the dashboard or under the "Account" section.

Look for something that says "API Key" or directly "User ID". This will be needed to authenticate API requests from your application.

Step 3: Create/Add a Service

Navigate to the "Email Services" section.

Click on “Add New Service” or if you already have a service set up, select it.

You can choose from various email service providers (like Gmail, Outlook, etc.). Follow the prompts to connect your email account.

After setting it up, EmailJS will assign a Service ID to this email connection. Note down this Service ID.

Step 4: Create an Email Template

Go to the "Email Templates" section.

Click on “Create new template” or choose an existing template.

Design your email template according to your needs. You can use variables like {{user_name}}, {{user_email}}, and {{message}} in your template. These placeholders will be replaced by the actual data from your form when an email is sent.

Once you have created the template, EmailJS will provide you with a Template ID for this specific template.

Step 5: Implement in Your React App

With these IDs in hand, you can integrate EmailJS in your React app. Here's a refresher on how to use these IDs in your application:

Lab7.js

```
import React, { useState } from 'react';
import emailjs from 'emailjs-com';

const Lab7 = () => {
    const [name, setName] = useState("");
    const [email, setEmail] = useState("");
    const [message, setMessage] = useState("");

    const sendEmail = (e) => {
        e.preventDefault();
      // Create a new object that contains dynamic template params
        const templateParams = {
            from_name: name,
            from_email: email,
            to_name: 'Bilahari',
            message: message,
        };

        emailjs.send('Your_Service_ID',            'Your_Template_ID',
templateParams, 'Your_User_ID')
            .then((result) => {
                console.log(result.text);
                alert('Email sent successfully!', result);
                setName('');
                setEmail('');
                setMessage('');
            }, (error) => {
                console.log(error.text);
                alert('Failed to send email.');
            });
    };

    return (
        <>
```

```
            <h3>Lab 7 Exercise</h3>
            <h4>Contact Us</h4>
            <form onSubmit={sendEmail} className='emailForm'>
                <label>Name:</label>
                <input
                    type="text"
                    placeholder="Your Name"
                    value={name}
                    onChange={(e) => setName(e.target.value)}
                />
                <label>Email:</label>
                <input
                    type="email"
                    placeholder="Your Email"
                    value={email}
                    onChange={(e) => setEmail(e.target.value)}
                />
                <label>Message:</label>
                <textarea
                    cols="30"
                    rows="10"
                    value={message}
                    onChange={(e) => setMessage(e.target.value)}
                />
                <input type="submit" value="Send Email" />
            </form>
        </>
    );
};

export default Lab7;
```