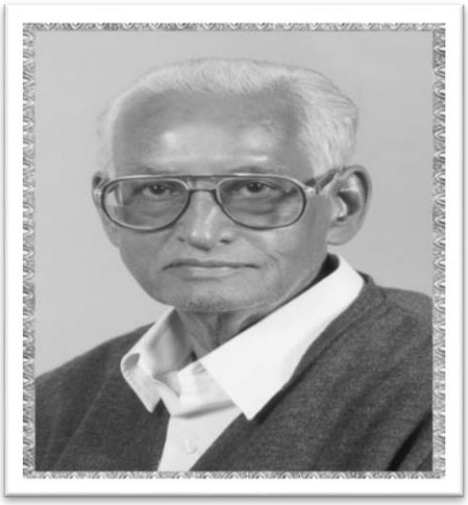
SHREE H. N. SHUKLA COLLEGE OF I.T. & MGMT.
(AFFILIATED TO SAURASHTRA UNIVERSITY)

.



Lt. Shree Chimanbhai Shukla

MSCIT SEM-2 REACTJS

Shree H.N.Shukla college2
vaishali nagar
Near Amrapali Under Bridge,
Raiya road
Rajkot
Ph No:-0281 2440478

Shree H.N.Shukla college3
vaishali nagar
Near Amrapali Under Bridge,
Raiya road
Rajkot
Ph No:-0281 2440478

Unit :3
Form Handling,Components and fragments

- **Event Handling:** Event Handling and Binding event handlers
- **Rendering:** Conditional Rendering and List Rendering, List and keys, Index as Key Anti-pattern
- **Introduction:** Basic form handling
- **Components:** Components Life Cycle Methods,
- Components Mounting Lifecycle methods,
- Components Updating Lifecycle methods, Pure

Event Handling and Binding event handlers reactjs

Here's a detailed explanation of event handling and binding event handlers in ReactJS:

1. Event Handling:

- **Purpose:** Respond to user interactions with elements in your React application.
- **Synthetic Events:** React creates its own event system with cross-browser compatibility.
- **Common Events:** Clicks, keypresses, form submissions, mouse movements, etc.

2. Binding Event Handlers:

- **Key Point:** Ensure this inside event handlers refers to the correct component instance.

- **Methods:**

  a. Arrow Functions: - Automatically inherit this from the enclosing scope. - Concise and often preferred for event handlers. jsx <button onClick={() => handleClick()}>Click me</button>

  b. Binding in Constructor (Class Components): - Use this.handleClick = this.handleClick.bind(this) in the constructor. jsx constructor(props) { super(props); this.handleClick = this.handleClick.bind(this); }

  c. Binding in Render (Less Efficient): - Bind directly in JSX using onClick={this.handleClick.bind(this)}. - May cause performance issues in large applications.

3. Accessing Event Data:

- **Event Object**: React passes a synthetic event object to handlers.
- **Properties:** Access information like target element, event type, coordinates, etc.

JavaScript

```
handleClick(event) {
  console.log(event.target); // Element that triggered the event
}
```

4. Best Practices:

- Prefer Arrow Functions: Cleaner syntax and this binding.
- Avoid Binding in Render: Potential performance overhead.
- Consider Event Delegation: For large lists or dynamic content, improve efficiency by handling events at a parent level.
- Use Libraries for Complex Forms: formik, react-hook-form, and others simplify form handling and validation.

Rendering: Conditional Rendering and List Rendering, List and keys, Index as Key Anti-pattern react js

In React, conditional rendering and list rendering are common patterns that allow you to dynamically control the content displayed in your components. Additionally, when working with lists, React requires each item in a list to have a unique "key" property to optimize rendering. However, using the index as the key is considered an anti-pattern in certain scenarios, and it's important to understand when to avoid it.

Conditional Rendering:
Conditional rendering is the practice of rendering different content based on certain conditions. It can be achieved using JavaScript expressions or ternary operators inside the JSX.

Example:

jsx

```
import React from 'react';

const ConditionalComponent = ({ isLoggedIn }) => {
  return (
    <div>
      {isLoggedIn ? (
        <p>Welcome, User!</p>
      ) : (
        <p>Please log in to access the content.</p>
      )}
    </div>
  );};
```

export default ConditionalComponent;

List Rendering:

List rendering involves mapping over an array and rendering each item in the array. This is commonly used when you have dynamic data to display.

Example:

jsx

```
import React from 'react';

const ListComponent = ({ items }) => {
  return (
    <ul>
      {items.map((item) => (
        <li key={item.id}>{item.name}</li>
      ))}
    </ul>
  );
};
```

export default ListComponent;

List and Keys:

When rendering lists in React, it's important to assign a unique key to each item in the list. Keys help React identify which items have changed, been added, or been removed. Keys should be stable, unique, and preferably associated with the data being rendered.

Example:

jsx

```
import React from 'react';

const ListComponent = ({ items }) => {
  return (
    <ul>
      {items.map((item) => (
        <li key={item.id}>{item.name}</li>
      ))}
    </ul>
  );
};
```

export default ListComponent;

Index as Key Anti-pattern:

While it's common to use the array index as the key when rendering a list, it can lead to performance issues and incorrect behavior in certain situations. This is especially problematic when the list is dynamic and items can be added or removed.

Anti-pattern Example:

jsx

```
import React from 'react';

const ListComponent = ({ items }) => {
  return (
    <ul>
      {items.map((item, index) => (
        // Using the index as the key (anti-pattern)
        <li key={index}>{item.name}</li>
      ))}
    </ul>
  );
};

export default ListComponent;
```

Why it's an Anti-pattern:

If the order of the items changes, or items are added or removed, React might not be able to correctly identify which item corresponds to which key.

It can lead to unnecessary re-renders and negatively impact performance.

Better Approach:

Use a unique identifier associated with the data, such as item.id, as the key.

jsx

```
{items.map((item) => (
  <li key={item.id}>{item.name}</li>
))}
```

Understanding when and how to use keys is crucial for optimizing the performance of your React applications, especially when dealing with dynamic lists.

Introduction: Basic form handling reactjs

Handling forms in React is a crucial part of building dynamic and interactive web applications. React provides a way to manage and handle forms by using state to keep track of form data and event handlers to respond to user input. Here's a basic example of form handling in React:

Let's create a simple React component with a form that allows users to input their name:

jsx

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

const BasicFormHandling = () => {
  // State to store the form data
  const [formData, setFormData] = useState({
    name: '',
  });

  // Event handler for input changes
  const handleInputChange = (e) => {
    const { name, value } = e.target;
    // Update the state with the new form data
    setFormData({
      ...formData,
      [name]: value,
    });
  };

  // Event handler for form submission
  const handleSubmit = (e) => {
    e.preventDefault();
    // Perform actions with the form data, such as making an API call or updating the state
    console.log('Form submitted with data:', formData);
  };

  return (
```

```
    <div>
      <h2>Basic Form Handling in React</h2>
      <form onSubmit={handleSubmit}>
        <label>
          Name:
          {/* Input field with an event handler */}
          <input
            type="text"
            name="name"
            value={formData.name}
            onChange={handleInputChange}
          />
        </label>
        <br />
        {/* Submit button */}
        <button type="submit">Submit</button>
      </form>
    </div>
  );
};

export default BasicFormHandling;
```

In this example:

We use the useState hook to create a state variable formData to store the form data. In this case, it's just the user's name.

The handleInputChange function is an event handler for input changes. It updates the state (formData) whenever the user types into the input field.

The handleSubmit function is an event handler for form submission. It prevents the default form submission behavior, allowing you to handle the form data in a custom way (e.g., making an API call).

The form JSX includes an input field for the user's name and a submit button. The input field has an onChange attribute that references the handleInputChange function, and the form has an onSubmit attribute that references the handleSubmit function.

This is a basic example, and you can expand on it by adding more form fields, validation, and additional logic based on your application's requirements.

Components:

Components Life Cycle Methods reactjs

In React, components have a lifecycle that includes various phases from creation to destruction. With the introduction of React Hooks, functional components can also utilize lifecycle-related features. Here's an overview of the lifecycle methods for class components and hooks used in functional components:

Class Components:

Mounting Phase:

constructor(): This method is called before the component is mounted. It's used for initializing state and binding methods.

static getDerivedStateFromProps(): Invoked right before rendering when new props or state are being received. It's used to update the state based on changes in props.

render(): The render method is responsible for rendering the component.

componentDidMount(): Called after the component has been rendered in the DOM. It's suitable for side effects such as fetching data from an API.

Updating Phase:

static getDerivedStateFromProps(): Similar to the mounting phase, it's invoked right before rendering when new props or state are being received.

s**houldComponentUpdate**(): Allows optimization by determining if the component should re-render. It can prevent unnecessary renders.

render(): Renders the component.

componentDidUpdate(): Called after the component is updated in the DOM. It's suitable for side effects related to the updated state or props.

Unmounting Phase:

componentWillUnmount(): Called just before the component is unmounted and destroyed. It's used for cleanup tasks like canceling network requests or clearing up subscriptions.

Functional Components with Hooks:

Mounting and Updating:

useEffect(): Combines componentDidMount, componentDidUpdate, and componentWillUnmount in functional components. It runs after every render and is used for side effects. The cleanup function in useEffect serves the same purpose as componentWillUnmount.

State Management:

useState(): Replaces the need for setState in class components for managing state in functional components.

Context:

useContext(): Allows functional components to subscribe to React context without introducing a nesting component.

Memoization:

useMemo(): Memoizes the result of a function, preventing unnecessary recalculations.

useCallback(): Memoizes a callback function, preventing it from being recreated on every render.

These lifecycle methods and hooks help you manage the state, perform side effects, and optimize the rendering process in your React components. Understanding these methods is crucial for building efficient and well-structured React applications. Keep in mind that with the introduction of React Hooks, many class component lifecycle methods are not used as frequently in modern React development.

Components Mounting Lifecycle methods

In React class components, the mounting lifecycle methods are invoked during the initial rendering of the component. These methods allow you to perform setup tasks, fetch data, and interact with the DOM. Here are the key mounting lifecycle methods in a React class component:

constructor():

The constructor method is the first method called when a component is created.
It is used for initializing state and binding event handlers.
Example:
jsx

```
constructor(props) {
  super(props);
  // Initialize state
  this.state = {
    data: [],
  };
  // Bind event handler
  this.handleClick = this.handleClick.bind(this);
}
```

static getDerivedStateFromProps():

Invoked right before rendering when new props or state are being received.
Used to update the state based on changes in props.
Example:
jsx

```
static getDerivedStateFromProps(nextProps, prevState) {
  if (nextProps.data !== prevState.data) {
    return {
      data: nextProps.data,
    };
  }
  return null;
}
```

render():

The render method is responsible for rendering the component.
It returns the React elements that make up the component's UI.
Example:
jsx

```
render() {
  return (
    <div>
      {/* Component's UI */}
    </div>
  );
}
```

componentDidMount():

Called after the component has been rendered in the DOM.
Used for performing side effects, such as fetching data from an API.
Example:
jsx

```
componentDidMount() {
  // Fetch data after the component has mounted
  fetchData()
    .then((data) => {
      this.setState({ data });
    })
    .catch((error) => {
      console.error('Error fetching data:', error);
    });
}
```

These mounting lifecycle methods are executed in the order listed above when a component is initially rendered. They provide opportunities to set up the initial state, handle props changes, render the component, and perform side effects after the component is mounted in the DOM. Keep in mind that with the introduction of React Hooks, the usage of class components and these lifecycle methods has become less common in modern React development. Hooks like useEffect in functional components are often used as an alternative.

Components Updating Lifecycle methods reactjs

In React class components, updating lifecycle methods are invoked when a component is re-rendered due to changes in its state or props. These methods allow you to control and perform actions during the updating phase of the component. Here are the key updating lifecycle methods in a React class component:

static getDerivedStateFromProps(nextProps, prevState):

Invoked right before rendering when new props or state are being received.
Used to update the state based on changes in props.

Example:

jsx

```
static getDerivedStateFromProps(nextProps, prevState) {
  if (nextProps.data !== prevState.data) {
    return {
      data: nextProps.data,
    };
  }
  return null;
}
```

shouldComponentUpdate(nextProps, nextState):

Called before rendering when new props or state are received.
Allows optimization by determining if the component should re-render.

Example:

jsx

```
shouldComponentUpdate(nextProps, nextState) {
  // Perform a check to determine if re-rendering is necessary
  return nextProps.value !== this.props.value || nextState.data !== this.state.data;
}
```

render():

The render method is responsible for rendering the component.
It returns the React elements that make up the component's UI.
getSnapshotBeforeUpdate(prevProps, prevState):

Invoked right before the most recently rendered output is committed to the DOM.
Used for capturing information from the DOM before it potentially changes.

Example:

jsx

```
getSnapshotBeforeUpdate(prevProps, prevState) {
  if (prevProps.list.length < this.props.list.length) {
    // Scroll position before the update
    return this.scrollRef.scrollTop;
  }
  return null;
}
componentDidUpdate(prevProps, prevState, snapshot):
```

Called after the component has been updated in the DOM.
Used for performing side effects related to the updated state or props.

Example:

jsx

```
componentDidUpdate(prevProps, prevState, snapshot) {
  if (snapshot !== null) {
    // Adjust scroll position after the update
    this.scrollRef.scrollTop = snapshot;
  }
}
```

These updating lifecycle methods provide opportunities to handle changes in state or props, control the re-rendering process, and perform side effects after the component has been updated in the DOM. Keep in mind that with the introduction of React Hooks, the usage of class components and these lifecycle methods has become less common in modern React development. Hooks like useEffect in functional components are often used as an alternative.

Pure:Components reactjs

In React, a "Pure Component" is a type of component that extends the React.PureComponent class instead of the regular React.Component class. The key feature of a pure component is that it performs a shallow comparison of its props and state to determine whether it should re-render. If the props and state haven't changed, a pure component prevents unnecessary renders, potentially improving performance.

Here's a basic example of a pure component:

Jsx

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

class PureExample extends PureComponent {
  render() {
    return (
      <div>
        <h2>Pure Component Example</h2>
        <p>Props value: {this.props.value}</p>
      </div>
    );
  }
}

export default PureExample;
```

Key points about pure components:

Shallow Comparison:

Pure components implement a shallow comparison of props and state.
This means that it checks if the references of the props and state objects are the same as in the previous render.
Automatic shouldComponentUpdate():

A pure component automatically implements the shouldComponentUpdate method with a shallow prop and state comparison.
It returns false if the props and state have not changed, preventing unnecessary renders.

Performance Optimization:

Pure components are useful in scenarios where you want to optimize performance by avoiding unnecessary renders.
They are particularly beneficial when dealing with large lists or datasets.
Example usage of a pure component in a parent component:

jsx

```
import React, { useState } from 'react';
import PureExample from './PureExample';

const ParentComponent = () => {
  const [value, setValue] = useState(0);

  const handleClick = () => {
    setValue(value + 1);
  };

  return (
    <div>
      <button onClick={handleClick}>Increment</button>
      <PureExample value={value} />
    </div>
  );
};

export default ParentComponent;
```

In this example, even if the parent component re-renders due to a state change, the pure component will only re-render if the value prop has changed. This can be especially beneficial in scenarios where the rendering of the pure component is resource-intensive.

Fragments react js

In React, a fragment is a lightweight way to group multiple elements without introducing an additional parent container in the DOM. Fragments don't create a new DOM element; they are like a wrapper that doesn't affect the DOM structure. Fragments were introduced to address the issue of unnecessary div wrappers that could be added when you want to return multiple elements from a component.

Here's how you can use fragments in React:

Using the <React.Fragment> syntax:

jsx

```
import React from 'react';

const MyComponent = () => {
  return (
    <React.Fragment>
      <h1>Hello</h1>
      <p>This is a paragraph.</p>
    </React.Fragment>
  );
};

export default MyComponent;
```

Short syntax using <> and </>:

In modern React versions, you can use the shorthand syntax using empty angle brackets (<> and </>). This is equivalent to using <React.Fragment>.

jsx

```
import React from 'react';

const MyComponent = () => {
  return (
    <>
      <h1>Hello</h1>
      <p>This is a paragraph.</p>
```

```
    </>
  );
};

export default MyComponent;
```

Key Points:

No Extra DOM Element:

Fragments do not create an additional DOM element. They allow you to group elements without introducing unnecessary wrappers.

Key Usage Scenarios:

Particularly useful when you need to return multiple elements from a component, for example, inside a map function.

jsx

```
const ListComponent = () => {
  const items = ['Item 1', 'Item 2', 'Item 3'];

  return (
    <>
      <h2>List of Items:</h2>
      <ul>
        {items.map((item, index) => (
          <li key={index}>{item}</li>
        ))}
      </ul>
    </>
  );
};
```

Keys and Attributes:

When using fragments in a list, make sure to provide a unique key to each element.

jsx

```
const MyComponent = () => {
  return (
    <>
      <p key="paragraph1">First paragraph</p>
      <p key="paragraph2">Second paragraph</p>
    </>
  );
};
```

Fragments provide a clean way to structure your JSX when you need to return multiple elements without introducing unnecessary container elements in the DOM. They help maintain a clean and semantic structure in your React components.