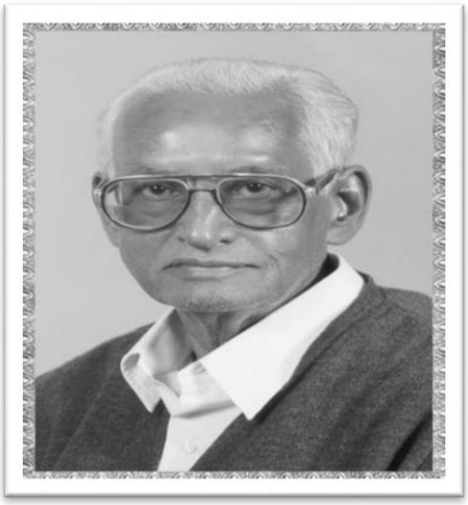
SHREE H. N. SHUKLA COLLEGE OF I.T. & MGMT.
(AFFILIATED TO SAURASHTRA UNIVERSITY)

.



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 :4
Memo, Refs,Props andContext

- **Memo**
- **Introduction to Refs:** Refs, Refs with Class Components, Forwarding Refs and Portals
- **Components:** Higher Order Components
- **Props Again!**: Rendering Props and Context
- **HTTP:** HTTP and React, GET and React, POST and React.

Memo

In React, the memo function is a higher-order component (HOC) that you can use to memoize functional components. Memoization is a technique to optimize rendering performance by preventing unnecessary re-renders of components. When a component is wrapped with memo, React will only re-render the component if its props have changed.

Here's how you can use memo in React:

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

// Functional component that doesn't use memo
const MyComponent = ({ prop1, prop2 }) => {
  console.log('Rendering MyComponent');
  return (
    <div>
      <p>Prop 1: {prop1}</p>
      <p>Prop 2: {prop2}</p>
```

```
    </div>
  );
};

// Wrapping MyComponent with memo
const MemoizedComponent = memo(MyComponent);

// Usage
const ParentComponent = () => {
  const [prop1, setProp1] = React.useState('Value 1');
  const [prop2, setProp2] = React.useState('Value 2');

  return (
    <div>
      <MemoizedComponent prop1={prop1} prop2={prop2} />
      <button onClick={() => setProp1('New Value 1')}>Change Prop 1</button>
      <button onClick={() => setProp2('New Value 2')}>Change Prop 2</button>
    </div>
  );
    };
```

In React, the memo function is a higher-order component (HOC) that you can use to memoize functional components. Memoization is a technique to optimize rendering performance by preventing unnecessary re-renders of components. When a component is wrapped with memo, React will only re-render the component if its props have changed.

Here's how you can use memo in React:

Javascript

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

// Functional component that doesn't use memo
const MyComponent = ({ prop1, prop2 }) => {
  console.log('Rendering MyComponent');
  return (
    <div>
      <p>Prop 1: {prop1}</p>
      <p>Prop 2: {prop2}</p>
    </div>
  );
};

// Wrapping MyComponent with memo
const MemoizedComponent = memo(MyComponent);

// Usage
const ParentComponent = () => {
  const [prop1, setProp1] = React.useState('Value 1');
  const [prop2, setProp2] = React.useState('Value 2');

  return (
    <div>
      <MemoizedComponent prop1={prop1} prop2={prop2} />
      <button onClick={() => setProp1('New Value 1')}>Change Prop 1</button>
      <button onClick={() => setProp2('New Value 2')}>Change Prop 2</button>
    </div>
  );
};
```

In this example:

MyComponent is a functional component that receives prop1 and prop2 as props.

MemoizedComponent is created by wrapping MyComponent with the memo function. This memoizes MyComponent and ensures that it only re-renders if its props change.

In the ParentComponent, MemoizedComponent is used and passed prop1 and prop2. There are buttons that update the state, causing the props to change. However, you'll notice that even when you click the buttons, the console log inside MyComponent only logs once, showing that it's not re-rendering on every state change due to memoization.

Memoization can significantly improve the performance of your React application, especially when dealing with complex components or large lists where unnecessary re-renders can occur. However, it's essential to use memoization judiciously, as it can also introduce complexity and potential bugs if used incorrectly.

Introduction to Refs: Refs, Refs with Class Components, Forwarding Refs and Portals

Refs in React are a feature that allows you to access and interact with DOM elements or React components directly. They provide a way to reference a specific element or component imperatively rather than declaratively. Refs are commonly used for managing focus, triggering imperative animations, integrating with third-party DOM libraries, and accessing DOM measurements.

There are several aspects of working with refs in React:

1. Refs with DOM Elements:

You can create a ref using **React.createRef()** and attach it to a DOM element in a component. This allows you to directly interact with the underlying DOM node.

Javascript Example

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

class MyComponent extends Component {
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }

  render() {
    return <div ref={this.myRef}>Hello World</div>;
  }
}
```

2. Refs with Class Components:

In class components, you can also use callback refs, which provide more flexibility, especially when dealing with multiple elements or dynamically created elements.

Javascript Example

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myRef = null;
  }

  setRef = (ref) => {
    this.myRef = ref;
  };

  render() {
    return <div ref={this.setRef}>Hello World</div>;
  }
}
```

3. Forwarding Refs:

Forwarding refs allows components to pass refs through to their children. This is useful when you're building a reusable component that needs to access the underlying DOM node or component instance.

Javascript Example

```
const FancyButton = React.forwardRef((props, ref) => (
  <button ref={ref} className="FancyButton">
    {props.children}
  </button>
));

// Now you can use FancyButton and pass a ref to it
const ref = React.createRef();
<FancyButton ref={ref}>Click me</FancyButton>;
```

Portals:

Portals provide a way to render children into a DOM node that exists outside the DOM hierarchy of the parent component. This is useful for scenarios like modals, tooltips, and popovers, where you want to render content outside the normal DOM flow.

Javascript Example

```
import React from 'react';
import ReactDOM from 'react-dom';

const modalRoot = document.getElementById('modal-root');

class Modal extends React.Component {
  constructor(props) {
    super(props);
    this.el = document.createElement('div');
  }

  componentDidMount() {
    modalRoot.appendChild(this.el);
  }

  componentWillUnmount() {
    modalRoot.removeChild(this.el);
  }

  render() {
    return ReactDOM.createPortal(this.props.children, this.el);
  }
}

// Usage
<Modal>
  <div>Modal Content</div>
</Modal>;
```

These are some of the key aspects of using refs in React, including working with DOM elements, class components, forwarding refs, and portals. Refs are a powerful tool in React, but they should be used sparingly and judiciously, as direct DOM manipulation can bypass React's declarative programming model and lead to harder-to-maintain code.

Components: Higher Order Components

In React.js, Higher Order Components (HOCs) are a powerful pattern used for code reuse, abstraction, and separation of concerns. They are functions that take a component and return a new component with extended or modified functionality. HOCs allow you to enhance components with additional props, state, or behavior without modifying the original component itself.

Here's a basic example of a Higher Order Component in React.js:

Example

```
import React from 'react';

// Define a Higher Order Component
const withLogger = (WrappedComponent) => {
  // Define a new component
  class WithLogger extends React.Component {
    componentDidMount() {
      console.log(`Component ${WrappedComponent.name} mounted`);
    }

    render() {
      // Render the original component with any props passed to it
      return <WrappedComponent {...this.props} />;
    }
  }

  return WithLogger;
};

// Define a regular component
class MyComponent extends React.Component {
  render() {
    return <div>Hello, World!</div>;
  }
}
```

```
// Enhance MyComponent with withLogger HOC
const MyComponentWithLogger = withLogger(MyComponent);

// Usage
ReactDOM.render(<MyComponentWithLogger/>, document.getElementById('root'));
```

In this example:

withLogger is a Higher Order Component that takes a component WrappedComponent as an argument.

Inside withLogger, a new component WithLogger is defined, which adds logging functionality to the lifecycle method componentDidMount.

WithLogger renders the original WrappedComponent with all its props passed down.

MyComponent is a regular component.

MyComponentWithLogger is the result of enhancing MyComponent with the withLogger HOC.

Higher Order Components are widely used in React.js for tasks like code reusability, state management, and abstraction of complex logic. They enable developers to compose components in a modular and flexible way. However, with the introduction of React Hooks, some patterns and use cases traditionally handled by HOCs can now be achieved using Hooks like useEffect and useState.

Props Again!: Rendering Props and Context

Rendering Props and Context are two powerful techniques in React.js for passing data from a parent component to its descendants. Let's discuss each one:

Rendering Props:

Rendering Props is a pattern where a component accepts a function as a prop, which it then calls to render content. This pattern allows for greater flexibility and reusability, as it enables components to render content based on the logic provided by the parent component.

Here's an example of how Rendering Props can be used:

```
import React from 'react';

// Parent component
class ParentComponent extends React.Component {
  render() {
    return (
      <div>
        {/* Passing a function as a prop */}
        <ChildComponent render={(name) => <div>Hello, {name}!</div>} />
      </div>
    );
  }
}

// Child component
class ChildComponent extends React.Component {
  render() {
    // Calling the function passed as a prop
    return <div>{this.props.render("World")}</div>;
  }
}
```

```
// Usage
    ReactDOM.render(<ParentComponent />, document.getElementById('root'));
```

In this example, ParentComponent passes a function as the render prop to ChildComponent. ChildComponent then calls this function with the argument "World" to render the content.

Context:

Context provides a way to pass data through the component tree without having to pass props down manually at every level. It's particularly useful for passing data that is needed by many components at different levels of the component tree.

Here's a basic example of using Context:

```
import React from 'react';

// Create a context
const MyContext = React.createContext();

// Parent component that provides the context value
class ParentComponent extends React.Component {
  render() {
    return (
      <MyContext.Provider value="World">
        <ChildComponent />
      </MyContext.Provider>
    );
  }
}

// Child component that consumes the context value
class ChildComponent extends React.Component {
  static contextType = MyContext;

  render() {
    return <div>Hello, {this.context}!</div>;
  }
}
```

// Usage
ReactDOM.render(<ParentComponent />, document.getElementById('root'));

In this example, ParentComponent provides the context value "World" using MyContext.Provider. ChildComponent consumes this context value using this.context.

Rendering Props and Context are both powerful techniques for passing data in React.js applications, and the choice between them depends on the specific use case and requirements of your application.

HTTP: HTTP and React, GET and React, POST and React.

In React.js, you can perform HTTP requests, such as GET and POST, using various methods and libraries. Two popular libraries for making HTTP requests in React are Axios and the built-in Fetch API. Let's explore how you can use these libraries to perform GET and POST requests in a React application.

GET Request with Axios:

Axios is a promise-based HTTP client for the browser and Node.js. To perform a GET request in React using Axios, you typically install Axios via npm or yarn and then use it in your components.

First, install Axios:

npm install axios

Then, you can make a GET request in a React component like this:

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

function App() {
  const [data, setData] = useState(null);

  useEffect(() => {
    axios.get('https://api.example.com/data')
      .then(response => {
        setData(response.data);
      })
      .catch(error => {
        console.error('Error fetching data:', error);
      });
  }, []);

  return (
    <div>
      {data ? (
        <div>Data: {JSON.stringify(data)}</div>
      ) : (
```

```
      <div>Loading...</div>
    )}
   </div>
  );
}

export default App;
```

In React.js, you can perform HTTP requests, such as GET and POST, using various methods and libraries. Two popular libraries for making HTTP requests in React are Axios and the built-in Fetch API. Let's explore how you can use these libraries to perform GET and POST requests in a React application.

GET Request with Axios:

Axios is a promise-based HTTP client for the browser and Node.js. To perform a GET request in React using Axios, you typically install Axios via npm or yarn and then use it in your components.

First, install Axios:

npm install axios

Then, you can make a GET request in a React component like this:

javascript

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

function App() {
  const [data, setData] = useState(null);

  useEffect(() => {
    axios.get('https://api.example.com/data')
      .then(response => {
        setData(response.data);
      })
      .catch(error => {
        console.error('Error fetching data:', error);
      });
  }, []);
```

```
  return (
    <div>
      {data ? (
        <div>Data: {JSON.stringify(data)}</div>
      ) : (
        <div>Loading...</div>
      )}
    </div>
  );
}

export default App;
```

POST Request with Fetch API:

The Fetch API is built into modern browsers and provides an interface for fetching resources across the network. You can use it to perform POST requests in React.

Here's an example of making a POST request with the Fetch API in a React component:

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

function App() {
  const [responseData, setResponseData] = useState(null);

  const postData = async () => {
    try {
      const response = await fetch('https://api.example.com/postData', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json'
        },
        body: JSON.stringify({ key: 'value' }) // Replace with your data
      });
      const data = await response.json();
      setResponseData(data);
    } catch (error) {
      console.error('Error posting data:', error);
    }
  };

  return (
```

```
    <div>
      <button onClick={postData}>Send POST Request</button>
      {responseData && (
        <div>Response: {JSON.stringify(responseData)}</div>
      )}
    </div>
  );
}

export default App;
```

In React.js, you can perform HTTP requests, such as GET and POST, using various methods and libraries. Two popular libraries for making HTTP requests in React are Axios and the built-in Fetch API. Let's explore how you can use these libraries to perform GET and POST requests in a React application.

GET Request with Axios:

Axios is a promise-based HTTP client for the browser and Node.js. To perform a GET request in React using Axios, you typically install Axios via npm or yarn and then use it in your components.

First, install Axios:

bash

```
npm install axios
```

Then, you can make a GET request in a React component like this:

javascript

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

function App() {
  const [data, setData] = useState(null);

  useEffect(() => {
    axios.get('https://api.example.com/data')
      .then(response => {
        setData(response.data);
```

```
    })
    .catch(error => {
      console.error('Error fetching data:', error);
    });
  }, []);

  return (
    <div>
      {data ? (
        <div>Data: {JSON.stringify(data)}</div>
      ) : (
        <div>Loading...</div>
      )}
    </div>
  );
}
```

export default App;

POST Request with Fetch API:

The Fetch API is built into modern browsers and provides an interface for fetching resources across the network. You can use it to perform POST requests in React.

Here's an example of making a POST request with the Fetch API in a React component:

javascript

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

function App() {
  const [responseData, setResponseData] = useState(null);

  const postData = async () => {
    try {
      const response = await fetch('https://api.example.com/postData', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json'
        },
        body: JSON.stringify({ key: 'value' }) // Replace with your data
      });
      const data = await response.json();
```

```
      setResponseData(data);
    } catch (error) {
      console.error('Error posting data:', error);
    }
  };

  return (
    <div>
      <button onClick={postData}>Send POST Request</button>
      {responseData && (
        <div>Response: {JSON.stringify(responseData)}</div>
      )}
    </div>
  );
}

export default App;
```

In these examples, axios.get and fetch are used to make GET and POSTrequests,respectively. Upon successful responses, the retrieved data is stored in React state (data or responseData) and then displayed in the component. Additionally, error handling is implemented to handle any network or API errors that may occur during the request.