MUHILAN ORG.

REACT JS E-BOOK

DREAM BIG,
DESIGN SMART..!

+91 8838299264

muhilan6601@outlook.com

Table of Contents

React

React is a JavaScript library for building user interfaces, primarily for web applications. It was developed by Facebook and released as open-source in 2013. React focuses on creating reusable UI components and managing their state efficiently, making it popular for building complex and interactive user interfaces.

Introduction

React is a popular JavaScript library for building user interfaces, particularly for single-page applications and large-scale web applications. Developed by Facebook, React allows developers to create interactive and dynamic UI components efficiently. Here's an introduction to React and its key concepts:

1. Component-Based Architecture:

React is built around the concept of reusable UI components. A React application is composed of multiple components, each responsible for rendering a part of the UI.

Components are self-contained and can be composed together to build complex UIs. They encapsulate both the UI elements and the logic associated with them.

2. Virtual DOM:

React uses a virtual DOM (Document Object Model) to efficiently update the UI. The virtual DOM is a lightweight representation of the actual DOM, maintained by React.

When the state of a component changes, React compares the virtual DOM with the previous version and only updates the necessary parts of the actual DOM. This approach improves performance by minimizing DOM manipulation.

3. JSX (JavaScript XML):

React introduces JSX, a syntax extension for JavaScript that allows you to write HTML-like code directly within JavaScript.

JSX makes it easier to describe the UI components' structure and improves code readability. It's then transformed into regular JavaScript by tools like Babel before being executed in the browser.

4. Unidirectional Data Flow:

React follows a unidirectional data flow model, where data flows from parent to child components through props (properties).

Changes to the data are propagated downwards through the component hierarchy, and child components cannot directly modify the data passed to them. Instead, they communicate changes back to their parent components through callbacks.

5. Declarative Syntax:

React promotes a declarative approach to building UIs, where you describe what you want the UI to look like based on the current state, rather than imperatively specifying how to change the UI in response to user actions.

Declarative code is often more concise, easier to understand, and less error-prone than imperative code.

6. React Hooks:

Introduced in React 16.8, hooks are functions that allow you to use state and other React features in function components, instead of having to use class components.

Hooks provide a more straightforward and composable way to manage component state, side effects, and lifecycle events.

7. Reusable Components:

React encourages the development of reusable and composable UI components, which can be shared and reused across different parts of the application.

This modular approach promotes code reusability, maintainability, and scalability, as components can be easily composed to create complex UIs.

React's simplicity, performance, and robust ecosystem of libraries and tools have made it a popular choice for front-end development. Whether you're building a simple web application or a complex enterprise-level application, React provides the flexibility and power to create modern, interactive user interfaces.

React Getting Started

To get started with React, you'll need to set up your development environment and create a new React project. Here's a step-by-step guide to help you get started:

1. Install Node.js and npm:

React applications are typically built using Node.js and npm (Node Package Manager). If you haven't already, download and install Node.js from the official website: Node.js.

npm is included with Node.js, so you don't need to install it separately.

2. Create a New React Project:

Once Node.js is installed, you can create a new React project using the create-react-app tool. Open your terminal or command prompt and run the following command:

luaCopy code

```
npx create-react-app my-react-app
```

Replace **my-react-app** with the name you want for your project. This command will create a new directory with the specified name and set up a basic React project structure for you.

3. Navigate to the Project Directory:

After the project is created, navigate to the project directory using the **cd** command:

bashCopy code

```
cd my-react-app
```

4. Start the Development Server:

Once you're inside the project directory, you can start the development server by running:

sqlCopy code

```
npm start
```

This command will start the development server and open your default web browser to view your React application running locally.

5. Explore the Project Structure:

Inside your project directory, you'll find the following structure:

perlCopy code

```
my-react-app/
├── node_modules/       # Dependencies installed via npm
├── public/             # Public assets and index.html
├── src/                # React application source code
│   ├── App.css         # CSS styles for App component
│   ├── App.js          # Main component
│   ├── App.test.js     # Test file
│   ├── index.css       # Global styles
│   ├── index.js        # Entry point
│   └── logo.svg        # Sample SVG logo
├── .gitignore          # Gitignore file
├── package-lock.json   # npm package lock file
├── package.json        # npm package configuration
└── README.md           # Project readme file
```

The **src** directory contains the main source code of your React application. The **App.js** file is the main component, and it's where you'll start building your application.

6. Start Building Your React App:

You can now start building your React application by editing the source files in the **src** directory.

Open **App.js** in your code editor and start modifying the component to create your UI.

7. Learn React:

React has a rich ecosystem of documentation, tutorials, and community resources to help you learn and master the library.

The official React documentation is a great place to start: React Documentation.

By following these steps, you'll have a basic React project set up and running locally on your development machine. You can then begin exploring React's features and building your own applications.

Upgrade React

To upgrade an existing React project to a newer version of React, you'll need to follow these general steps:

1. **Check Current Version:**

   Determine which version of React your project is currently using. You can find this information in your `package.json` file under the `dependencies` section.

2. **Check for Compatibility:**

   Before upgrading, check the React release notes and documentation for any breaking changes or deprecations between your current version and the version you want to upgrade to. Make sure there are no major compatibility issues that could affect your project.

3. **Update React and React DOM:**

   Once you've verified compatibility, update React and React DOM to the desired version. You can do this using npm:

graphqlCopy code

```
npm install react@latest react-dom@latest
```

   Replace `latest` with the specific version you want to upgrade to if you're not using the latest version.

4. **Update Other Dependencies:**

   Sometimes, upgrading React may require updating other dependencies or packages in your project to ensure compatibility. Check the React documentation and release notes for any additional dependencies that may need updating.

5. **Check for Deprecated Features:**

   After updating React, review your codebase for any deprecated features or APIs that have been removed in the new version. Update your code to use the recommended alternatives.

6. **Test Your Application:**

   Test your application thoroughly after upgrading to ensure that everything is working as expected. Pay special attention to any areas of your application that use React-specific features or APIs.

7. **Fix Compatibility Issues:**

   If you encounter any compatibility issues or errors after upgrading, troubleshoot and fix them accordingly. This may involve updating third-party libraries, modifying code, or implementing workarounds.

8. Update Documentation:

Update your project's documentation to reflect the changes and new features introduced in the upgraded version of React. This will help other developers understand how to work with the updated codebase.

9. Commit Changes:

Once you've completed the upgrade process and tested your application, commit your changes to version control (e.g., Git) to track the updates and make it easier to roll back if necessary.

10. Monitor for Updates:

Keep an eye on future React releases and updates to stay informed about new features, bug fixes, and improvements. Regularly updating your project ensures that you're using the latest stable version of React and staying up-to-date with the latest developments in the ecosystem.

By following these steps, you can safely upgrade your React project to a newer version while ensuring compatibility and minimizing the risk of introducing bugs or breaking changes.

React ES6

React, being a JavaScript library, leverages many features introduced in ECMAScript 6 (ES6), also known as ECMAScript 2015. ES6 introduced significant enhancements to the JavaScript language, making it more powerful and expressive. Here's how ES6 features are commonly used in React development:

1. Arrow Functions:

Arrow functions provide a more concise syntax for defining functions, especially for inline callbacks and event handlers.

Example:

```
const MyComponent = () => {
   return <button onClick={() => console.log('Clicked')}>
   Click Me</button>;
};
```

2. Classes:

ES6 classes offer a more structured way to define React components, replacing the older **createClass** method.

Example:

```
class MyComponent extends React.Component {
   render() {
   return <h1>Hello, World!</h1>;
   }
}
```

3. Template Literals:

Template literals allow for easier string interpolation and multi-line strings in JSX templates.

Example:

```
const name = 'John';
return <h1>Hello, {`My name is ${name}`}</h1>;
```

4. Destructuring:

Destructuring assignment simplifies the extraction of values from objects and arrays, often used with props and state.

Example:

```
const { name, age } = this.props;
```

5. Spread Syntax:

The spread syntax (...) allows for the expansion of iterable objects like arrays and objects, commonly used for passing props or merging objects.

Example:

```
const props = { name: 'John', age: 30 };
return <MyComponent {...props} />;
```

6. Modules:

ES6 introduced a standardized module system (**import** and **export** keywords), which is commonly used for organizing and importing/exporting React components and utilities.

Example:

```
// Component.js
import React from 'react';
const MyComponent = () => <h1>Hello, World!</h1>;
export default MyComponent;

// App.js
import React from 'react';
import MyComponent from './Component';
```

7. Let and Const:

let and **const** provide block-scoped variables, which are commonly used in place of **var** for improved variable scoping and immutability.

Example:

```
let count = 0;
const maxCount = 10;
```

These are just a few examples of how ES6 features are commonly used in React development. Leveraging these features can lead to cleaner, more expressive, and more maintainable React code.

React ES6 Classes

In React, ES6 classes are commonly used to define components. ES6 classes provide a more structured and familiar syntax for defining components compared to the older **createClass** method. Here's how ES6 classes are used to define React components:

1. Class Declaration:

You can define a React component by creating a class that extends **React.Component**.

The class must have a **render()** method that returns the component's JSX structure.

Example:

```
import React from 'react';

class MyComponent extends React.Component {
render() {
   return <h1>Hello, World!</h1>;
}
}

export default MyComponent;
```

2. Constructor and State:

You can define a constructor in the class to initialize the component's state and bind event handlers.

The **super()** method must be called inside the constructor to initialize the parent class (**React.Component**).

Example:

```
javascriptCopy code
import React from 'react'; class MyComponent extends React.Component { render() { return
<h1>Hello, World!</h1>; } } export default MyComponent;
```

3. Props:

Props are passed to the component as parameters to the constructor and are accessible using **this.props**.

Example:

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

```
class Greeting extends React.Component {
render() {
   return <h1>Hello, {this.props.name}!</h1>;
}
}

export default Greeting;
```

4. Lifecycle Methods:

React class components have lifecycle methods that allow you to hook into component lifecycle events, such as mounting, updating, and unmounting.

Example:

```
import React from 'react';

class MyComponent extends React.Component {
componentDidMount() {
   console.log('Component mounted');
}

componentWillUnmount() {
   console.log('Component will unmount');
}

render() {
   return <h1>Hello, World!</h1>;
}
}

export default MyComponent;
```

Using ES6 classes to define React components offers a more modern and structured approach to building UIs, making it easier to manage component state, lifecycle events, and code organization.

React ES6 Arrow Functions

In React, ES6 arrow functions are commonly used for defining functional components, event handlers, and callback functions. Arrow functions provide a concise and more readable syntax compared to traditional function expressions, especially for inline functions and callbacks. Here's how ES6 arrow functions are used in React:

1. Functional Components:

Arrow functions can be used to define functional components, which are stateless components that only render UI based on props.

Example:

```
import React from 'react';

const MyComponent = () => {
return <h1>Hello, World!</h1>;
};

export default MyComponent;
```

2. Inline Event Handlers:

Arrow functions are often used for inline event handlers in JSX, providing a convenient way to define event handling logic directly within the component's render method.

Example:

```
import React from 'react';

const MyComponent = () => {
const handleClick = () => {
   console.log('Button clicked');
};

return <button onClick={handleClick}>Click Me</button>;
};

export default MyComponent;
```

3. Props with Arrow Functions:

Arrow functions are useful for passing props with inline functions, allowing you to pass arguments dynamically to event handlers or callback functions.

Example:

```
import React from 'react';

const Greeting = ({ name }) => {
const handleClick = () => {
   console.log(`Hello, ${name}!`);
};

return <button onClick={handleClick}>Greet {name}</button>;
};

export default Greeting;
```

4. Class Methods:

Arrow functions can be used to define class methods within React component classes. When using arrow functions for class methods, you don't need to manually bind **this** in the constructor.

Example:

```
import React from 'react';

class Counter extends React.Component {
state = {
   count: 0
};

handleClick = () => {
   this.setState({ count: this.state.count + 1 });
};

render() {
   return (
   <div>
      <h1>Counter: {this.state.count}</h1>
      <button onClick={this.handleClick}>Increment</button>
   </div>
   );
}
}

export default Counter;
```

Using arrow functions in React components can lead to more concise and readable code, especially for defining inline functions and event handlers. However, it's important to consider potential performance implications, especially when using arrow functions for class methods, as they create a new function instance on every render.

React ES6 Variables

In React, ES6 variables (const and let) are commonly used for defining and managing variables within components. ES6 introduced the **const** and **let** keywords for declaring variables, offering block-scoped variables that provide better control over variable scope and immutability. Here's how **const** and **let** variables are used in React:

1. Const Variables:

const is used to declare variables whose values cannot be reassigned after initialization.

It's commonly used for defining variables that represent immutable values or references to components.

Example:

```
import React from 'react';

const MyComponent = () => {
const message = 'Hello, World!';

return <h1>{message}</h1>;
};

export default MyComponent;
```

2. Let Variables:

let is used to declare variables that can be reassigned within the block scope in which they are defined.

It's commonly used for defining variables that may change their values during the component's lifecycle.

Example:

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

const MyComponent = () => {
let count = 0;

const handleClick = () => {
   count += 1;
   console.log(`Clicked ${count} times`);
};

return (
```

```
    <div>
    <h1>Counter: {count}</h1>
    <button onClick={handleClick}>Increment</button>
    </div>
);
};

export default MyComponent;
```

3. Destructuring with Const and Let:

const and **let** can be used with destructuring assignment to extract values from objects or arrays.

This is commonly used with props and state variables in React components.

Example:

```
import React from 'react';

const MyComponent = ({ name }) => {
const { age } = props;

let [count, setCount] = useState(0);

// Other code...
};

export default MyComponent;
```

4. Block Scoping:

Both **const** and **let** variables are block-scoped, meaning they are only accessible within the block (enclosed by curly braces) in which they are defined.

This helps prevent variable hoisting issues and makes code easier to understand and reason about.

Example:

```
import React from 'react';

const MyComponent = () => {
if (true) {
  const message = 'Hello, World!';
```

```
    console.log(message); // Output: "Hello, World!"
}
console.log(message); // Error: message is not defined
};

export default MyComponent;
```

Using **const** and **let** variables in React components helps create more predictable and maintainable code, ensuring that variables are scoped appropriately and their values are controlled effectively throughout the component's lifecycle.

React ES6 Array Methods

In React, ES6 array methods are commonly used for manipulating arrays of data within components. These methods provide powerful and concise ways to iterate, filter, map, and reduce arrays, making it easier to work with data in React components. Here are some commonly used ES6 array methods in React:

1. map():

The **map()** method creates a new array by applying a function to each element of the original array.

It's commonly used for rendering lists of data in React components.

Example:

```
import React from 'react';

const MyComponent = () => {
const numbers = [1, 2, 3, 4, 5];

return (
   <ul>
   {numbers.map((number) => (
      <li key={number}>{number}</li>
   ))}
   </ul>
);
};

export default MyComponent;
```

2. filter():

The **filter()** method creates a new array with all elements that pass a certain condition specified by a callback function.

It's commonly used for filtering data based on specific criteria.

Example:

```
import React from 'react';

const MyComponent = () => {
const numbers = [1, 2, 3, 4, 5];

const evenNumbers = numbers.filter((number) => number % 2 === 0);
```

```
return (
    <ul>
    {evenNumbers.map((number) => (
        <li key={number}>{number}</li>
    ))}
    </ul>
);
};

export default MyComponent;
```

3. reduce():

The **reduce()** method applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value.

It's commonly used for aggregating data or performing calculations on arrays.

Example:

```
import React from 'react';

const MyComponent = () => {
const numbers = [1, 2, 3, 4, 5];

const sum = numbers.reduce((total, current) => total + current, 0);

return <p>Sum: {sum}</p>;
};

export default MyComponent;
```

4. find():

The **find()** method returns the value of the first element in the array that satisfies a provided testing function.

It's commonly used for finding a specific item in an array.

Example:

```
import React from 'react';

const MyComponent = () => {
```

```
const users = [
    { id: 1, name: 'John' },
    { id: 2, name: 'Jane' },
    { id: 3, name: 'Doe' }
];

const user = users.find((user) => user.id === 2);

return <p>User: {user.name}</p>;
};

export default MyComponent;
```

These are just a few examples of how ES6 array methods can be used in React components to manipulate and process arrays of data efficiently. By leveraging these methods, you can write cleaner and more expressive code when working with arrays in your React applications.

React ES6 Destructuring

Destructuring in ES6 allows you to extract values from objects or arrays and assign them to variables in a more concise and readable way. In React, destructuring is commonly used for accessing props, state, and other objects within functional components. Here's how destructuring is used in React ES6:

1. Destructuring Props:

In functional components, props are passed as an object parameter. Destructuring allows you to extract individual props into separate variables for easier access.

Example:

```
import React from 'react';

const Greeting = ({ name }) => {
return <h1>Hello, {name}!</h1>;
};

export default Greeting;
```

2. Destructuring State:

In class components, state is accessed via **this.state**. Destructuring can be used within the **render()** method to extract state values into variables.

Example:

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

class Counter extends Component {
constructor(props) {
   super(props);
   this.state = { count: 0 };
}

render() {
   const { count } = this.state;
   return <h1>Count: {count}</h1>;
}
}

export default Counter;
```

3. Destructuring Arrays:

Destructuring can also be used with arrays to extract values based on their positions.

Example:

```
import React from 'react';

const MyComponent = () => {
const [first, second] = [1, 2];
console.log(first); // Output: 1
console.log(second); // Output: 2
return null;
};

export default MyComponent;
```

4. Destructuring in Function Arguments:

You can destructure objects or arrays directly within function arguments.

Example:

```
import React from 'react';

const Person = ({ name, age }) => {
return <p>Name: {name}, Age: {age}</p>;
};

export default Person;
```

5. Default Values:

Destructuring allows you to set default values for variables in case the destructured value is **undefined**.

Example:

```
import React from 'react';

const MyComponent = ({ message = 'Hello, World!' }) => {
return <h1>{message}</h1>;
};

export default MyComponent;
```

Destructuring in React helps to simplify code and make it more readable by reducing repetitive access to props, state, and other data structures. It's a powerful feature that can significantly improve the readability and maintainability of your React components.

React ES6 Spread Operator

In React, the ES6 spread operator (...) is a powerful feature used for manipulating arrays and objects. It allows you to easily copy, merge, and spread the elements of an array or the properties of an object into another array or object. The spread operator is commonly used in various scenarios within React components. Here's how the spread operator is used:

1. Copying Arrays:

The spread operator can be used to create a shallow copy of an array. This is useful when you want to modify an array without mutating the original.

Example:

```
const originalArray = [1, 2, 3];
const copiedArray = [...originalArray];
```

2. Merging Arrays:

The spread operator can be used to merge two or more arrays into a single array.

Example:

```
const array1 = [1, 2, 3];
const array2 = [4, 5, 6];
const mergedArray = [...array1, ...array2];
```

3. Copying Objects:

The spread operator can be used to create a shallow copy of an object. This is useful when you want to modify an object without mutating the original.

Example:

```
const originalObject = { name: 'John', age: 30 };
const copiedObject = { ...originalObject };
```

4. Merging Objects:

The spread operator can be used to merge two or more objects into a single object.

Example:

```
```

```
const object1 = { name: 'John' };
const object2 = { age: 30 };
const mergedObject = { ...object1, ...object2 };
```

5. Passing Props:

In React, the spread operator is commonly used to pass props to child components dynamically. This allows for cleaner and more concise code.

Example:

```
const MyComponent = (props) => {
  return <ChildComponent {...props} />;
};
```

6. Updating State:

When updating state in React, the spread operator is often used to merge the existing state with the new state.

Example:

```
this.setState({
  ...this.state,
  count: this.state.count + 1
});
```

The spread operator is a versatile tool in React development, providing a concise and efficient way to work with arrays and objects. It helps simplify code and improves readability, making it a valuable feature in React components.

React ES6 Modules

In React, ES6 modules are used to organize and manage the codebase by breaking it down into smaller, reusable modules. ES6 modules allow you to export functions, classes, or variables from one module and import them into another module where they are needed. This modular approach helps improve code maintainability, reusability, and scalability. Here's how ES6 modules are used in React:

1. Exporting from a Module:

To export functions, classes, or variables from a module, you use the **export** keyword followed by the declaration.

Example:

```
// utils.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;
```

2. Named Exports:

You can have multiple named exports in a single module by using the **export** keyword before each declaration.

Example:

```
// utils.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;
```

3. Default Export:

In addition to named exports, you can also have a default export in a module using the **export default** syntax.

Example:

```
// utils.js
const multiply = (a, b) => a * b;
export default multiply;
```

4. Importing into a Module:

To use functions, classes, or variables from another module, you use the **import** keyword followed by the module path and the names of the exports in curly braces (for named exports) or without braces (for default exports).

Example:

```
// App.js
import { add, subtract } from './utils';
import multiply from './utils';

console.log(add(2, 3)); // Output: 5
console.log(subtract(5, 2)); // Output: 3
console.log(multiply(2, 4)); // Output: 8
```

5. Renaming Imports:

You can rename named imports by using the **as** keyword followed by the new name.

Example:

```
// App.js
import { add as addition, subtract as subtraction } from './utils';

console.log(addition(2, 3)); // Output: 5
console.log(subtraction(5, 2)); // Output: 3
```

6. Importing All Exports:

You can import all named exports from a module using the *** as** syntax followed by a variable name.

Example:

```
// App.js
import * as utils from './utils';

console.log(utils.add(2, 3)); // Output: 5
console.log(utils.subtract(5, 2)); // Output: 3
```

ES6 modules provide a clean and efficient way to organize code in React applications, allowing for better separation of concerns and improved code maintainability. By leveraging modular architecture, you can easily manage dependencies, reuse code, and scale your React application effectively.

React ES6 Ternary Operator

In React, the ES6 ternary operator (`? :`) is commonly used for conditional rendering and conditional logic within JSX expressions. The ternary operator provides a concise and readable way to conditionally render different elements or values based on certain conditions. Here's how the ternary operator is used in React:

1. Conditional Rendering:

The ternary operator is often used to conditionally render different JSX elements based on a condition.

Example:

```
import React from 'react';

const Greeting = ({ isLoggedIn }) => {
return (
   <div>
   {isLoggedIn ? <p>Welcome, User!</p> : <p>Please log in.</p>}
   </div>
);
};

export default Greeting;
```

2. Conditional Styling:

You can use the ternary operator to conditionally apply CSS classes or inline styles based on certain conditions.

Example:

```
import React from 'react';

const Button = ({ isPrimary }) => {
return (
   <button style={{ backgroundColor: isPrimary ? 'blue' : 'gray' }}>
   {isPrimary ? 'Primary' : 'Secondary'}
   </button>
);
};

export default Button;
```

3. Conditional Logic:

The ternary operator can be used for conditional logic within JSX expressions to determine which value or element to render.

Example:

```
import React from 'react';

const Temperature = ({ temperature }) => {
return (
   <p>
   The water is {temperature >= 100 ? 'boiling' : 'not boiling'}.
   </p>
);
};

export default Temperature;
```

4. Conditional Rendering in JSX Attributes:

You can use the ternary operator within JSX attributes to conditionally set attribute values based on certain conditions.

Example:

```
import React from 'react';

const Input = ({ disabled }) => {
return (
   <input
   type="text"
   placeholder={disabled ? 'Disabled' : 'Enabled'}
   disabled={disabled}
   />
);
};

export default Input;
```

The ternary operator is a powerful tool for writing conditional logic and conditional rendering in React components. It helps keep the JSX code clean, concise, and readable, making it easier to understand and maintain your React applications.

React Render HTML

In React, you can render HTML elements using JSX syntax. JSX allows you to write HTML-like code directly within your JavaScript files, which gets compiled into regular JavaScript code that React can understand and render. Here's how you can render HTML elements in a React component:

Example:

```
import React from 'react';

const MyComponent = () => {
return (
   <div>
   <h1>Hello, World!</h1>
   <p>This is a paragraph.</p>
   <ul>
      <li>Item 1</li>
      <li>Item 2</li>
      <li>Item 3</li>
   </ul>
   </div>
);
};

export default MyComponent;
```

In the example above, we've defined a functional component **MyComponent** that returns JSX containing various HTML elements such as **div**, **h1**, **p**, and **ul** with **li** elements. When this component is rendered, React will create a corresponding DOM structure based on the JSX elements.

You can also embed JavaScript expressions within JSX by wrapping them in curly braces {}. This allows you to dynamically generate content based on variables or props:

Example:

```
import React from 'react';

const MyComponent = () => {
const name = 'John';
const greeting = `Hello, ${name}!`;

return (
   <div>
   <h1>{greeting}</h1>
   <p>{name} is learning React.</p>
   </div>
```

```
  );
  };

  export default MyComponent;
```

In this example, the **greeting** variable is dynamically generated using the **name** variable and embedded within the JSX code.

Remember that JSX is not HTML, even though it looks very similar. JSX is a syntax extension for JavaScript, and React components written in JSX are transformed into regular JavaScript function calls behind the scenes. This is why JSX allows you to write HTML-like code within your JavaScript files.

React JSX

JSX (JavaScript XML) is a syntax extension for JavaScript that allows you to write HTML-like code within your JavaScript files. It's commonly used with React to describe what the UI should look like. JSX makes React components more readable and easier to write by combining HTML-like syntax with JavaScript expressions. Here are some key points about JSX:

1. HTML-Like Syntax:

JSX looks similar to HTML but is actually closer to JavaScript. It allows you to write HTML elements and attributes directly within your JavaScript code.

Example:

```
const element = <h1>Hello, World!</h1>;
```

2. JavaScript Expressions:

JSX allows you to embed JavaScript expressions within curly braces `{}`. This allows you to dynamically generate content, compute values, or reference variables within JSX.

Example:

```
const name = 'John';
const element = <h1>Hello, {name}!</h1>;
```

3. Components as JSX Elements:

JSX elements can also represent React components. You can use custom components just like built-in HTML elements.

Example:

```
import CustomComponent from './CustomComponent';

const element = <CustomComponent />;
```

4. Attributes and Props:

JSX allows you to specify HTML attributes using camelCase naming convention. You can also pass props to components using attributes.

Example:

```

```

```
const element = <input type="text" className="input" onChange={handleChange} />;
```

5. Expression as Attribute Value:

You can use JavaScript expressions as attribute values within curly braces `{}`. This allows you to dynamically set attribute values based on variables or expressions.

Example:

```
const imageUrl = 'https://example.com/image.jpg';
const element = <img src={imageUrl} alt="Image" />;
```

6. Multiline JSX:

JSX expressions can span multiple lines. However, if the JSX expression is multiline, it should be wrapped in parentheses `()` to avoid automatic semicolon insertion issues.

Example:

```
const element = (
  <div>
  <h1>Hello, World!</h1>
  <p>This is a paragraph.</p>
  </div>
);
```

JSX simplifies the process of creating and working with React components by providing a familiar HTML-like syntax combined with the expressive power of JavaScript. It's an essential part of writing React applications and is widely adopted in the React ecosystem.

React Components

In React, components are the building blocks of a user interface. They are reusable, self-contained pieces of UI that encapsulate a specific functionality or visual aspect. React applications are typically composed of multiple components, each responsible for rendering a part of the UI. Components can be either functional components or class components. Here's an overview of React components:

1. Functional Components:

Functional components are simple JavaScript functions that accept props as arguments and return JSX to describe what the UI should look like.

They are preferred for presentational components that are mainly concerned with rendering UI based on props.

Example:

```
import React from 'react';

const Greeting = (props) => {
return <h1>Hello, {props.name}!</h1>;
};

export default Greeting;
```

2. Class Components:

Class components are ES6 classes that extend **React.Component** and have a **render()** method to return JSX.

They have additional features such as state and lifecycle methods, making them suitable for more complex components.

Example:

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

class Counter extends Component {
constructor(props) {
   super(props);
   this.state = { count: 0 };
}

render() {
   return <h1>Count: {this.state.count}</h1>;
}
}
```

```
export default Counter;
```

3. Props:

Props (short for properties) are used to pass data from parent components to child components. They are immutable and are passed down the component tree.

Example:

```
// Parent component
<Greeting name="John" />

// Child component
const Greeting = (props) => {
return <h1>Hello, {props.name}!</h1>;
};
```

4. State:

State is a data structure that represents the internal state of a component. It is mutable and can be updated using **setState()** method.

State is managed internally within a component and is not accessible from outside.

Example:

```
class Counter extends Component {
  constructor(props) {
  super(props);
  this.state = { count: 0 };
  }

  handleClick = () => {
  this.setState({ count: this.state.count + 1 });
  };

  render() {
  return (
    <div>
    <h1>Count: {this.state.count}</h1>
    <button onClick={this.handleClick}>Increment</button>
    </div>
  );
```

```
    }
}
```

5. Lifecycle Methods:

Class components have lifecycle methods that allow you to hook into various stages of a component's lifecycle, such as mounting, updating, and unmounting.

Example:

```
class MyComponent extends Component {
    componentDidMount() {
    // Called after the component is mounted
    }

    componentDidUpdate(prevProps, prevState) {
    // Called after the component updates
    }

    componentWillUnmount() {
    // Called before the component is unmounted
    }

    render() {
    return <h1>Hello, World!</h1>;
    }
}
```

React components allow you to break down your UI into smaller, reusable pieces, making your code more modular, maintainable, and easier to reason about. Components are at the core of React's declarative and composable nature.

React Class Components

React class components are ES6 classes that extend **React.Component** and have a **render()** method. They are used to define components with more complex logic, state management, and lifecycle methods. Class components are typically used when you need to manage state or use lifecycle methods. Here's how you can create and use class components in React:

1. Creating a Class Component:

You can create a class component by defining a JavaScript class that extends **React.Component** and implements a **render()** method.

Example:

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

class MyComponent extends Component {
render() {
  return <h1>Hello, World!</h1>;
}
}

export default MyComponent;
```

2. Adding State:

Class components can have state, which allows them to manage data that can change over time.

State is initialized in the constructor using **this.state** and can be accessed using **this.state**.

Example:

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

class Counter extends Component {
constructor(props) {
  super(props);
  this.state = { count: 0 };
}

render() {
  return (
  <div>
    <h1>Count: {this.state.count}</h1>
```

```
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
        Increment
        </button>
    </div>
    );
}
}

export default Counter;
```

3. Handling Events:

Class components can handle events by passing event handlers as props to JSX elements.

Inside event handlers, you can update the component's state using **this.setState()**.

Example:

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

class Button extends Component {
handleClick = () => {
    console.log('Button clicked!');
};

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

export default Button;
```

4. Using Lifecycle Methods:

Class components have lifecycle methods that allow you to hook into different stages of the component's lifecycle, such as mounting, updating, and unmounting.

Example:

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

class MyComponent extends Component {
componentDidMount() {
    console.log('Component mounted');
```

```
  }

  componentWillUnmount() {
    console.log('Component will unmount');
  }

  render() {
    return <h1>Hello, World!</h1>;
  }
  }

  export default MyComponent;
```

Class components are powerful and provide more features compared to functional components, such as state management and lifecycle methods. However, with the introduction of React hooks, functional components have become more capable and are now the preferred way of writing components in React.

React Props

In React, props (short for properties) are a way of passing data from parent components to child components. Props allow you to customize and configure child components by providing them with data and behavior. Props are immutable, meaning that child components cannot modify the props passed to them. Here's how you can work with props in React:

1. Passing Props:

You can pass props to a child component by adding attributes to the component's JSX tag. These attributes represent the data or behavior you want to pass.

Example:

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

const ParentComponent = () => {
return <ChildComponent name="John" age={30} />;
};

export default ParentComponent;
```

2. Accessing Props:

Inside the child component, you can access the props passed from the parent component via the function argument.

Example:

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

const ChildComponent = (props) => {
return (
   <div>
   <p>Name: {props.name}</p>
   <p>Age: {props.age}</p>
   </div>
);
};

export default ChildComponent;
```

3. Destructuring Props:

You can use object destructuring to extract individual props from the props object for easier access.

Example:

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

const ChildComponent = ({ name, age }) => {
return (
   <div>
   <p>Name: {name}</p>
   <p>Age: {age}</p>
   </div>
);
};

export default ChildComponent;
```

4. Default Props:

You can specify default values for props using the **defaultProps** property. If a prop is not provided, React will use the default value.

Example:

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

const ChildComponent = ({ name, age }) => {
return (
   <div>
   <p>Name: {name}</p>
   <p>Age: {age}</p>
   </div>
);
};

ChildComponent.defaultProps = {
name: 'Anonymous',
age: 18
};

export default ChildComponent;
```

Props are a fundamental concept in React that allows you to create reusable and configurable components. They enable you to build flexible and dynamic UIs by passing data between components in a unidirectional flow.

React Events

In React, event handling is similar to handling events in HTML, but with some differences due to JSX syntax and React's synthetic event system. React provides a straightforward way to handle events such as onClick, onChange, onSubmit, etc., using JSX syntax. Here's how you can handle events in React:

1. Event Handling Syntax:

You can handle events by passing event handler functions as props to JSX elements.

Example:

```
import React from 'react';

const MyComponent = () => {
const handleClick = () => {
  console.log('Button clicked!');
};

return <button onClick={handleClick}>Click Me</button>;
};

export default MyComponent;
```

2. Event Object:

React wraps the native DOM events in its own synthetic event system to ensure consistent behavior across different browsers.

You can access the event object passed to event handlers as the first argument.

Example:

```
import React from 'react';

const MyComponent = () => {
const handleClick = (event) => {
  console.log('Button clicked!', event.target);
};

return <button onClick={handleClick}>Click Me</button>;
};

export default MyComponent;
```

3. Preventing Default Behavior:

You can prevent the default behavior of events using the **preventDefault()** method on the event object.

Example:

```
import React from 'react';

const MyComponent = () => {
const handleSubmit = (event) => {
   event.preventDefault();
   console.log('Form submitted!');
};

return (
   <form onSubmit={handleSubmit}>
   <button type="submit">Submit</button>
   </form>
);
};

export default MyComponent;
```

4. Passing Parameters:

If you need to pass additional parameters to an event handler, you can use an arrow function or **Function.prototype.bind**.

Example with arrow function:

```
import React from 'react';

const MyComponent = () => {
const handleClick = (id) => {
   console.log(`Button clicked with id: ${id}`);
};

return <button onClick={() => handleClick(1)}>Click Me</button>;
};

export default MyComponent;
```

5. Class Component Event Handling:

In class components, event handlers are defined as methods of the component class.

Example:

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

class MyComponent extends Component {
handleClick = () => {
   console.log('Button clicked!');
};

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

export default MyComponent;
```

React's event handling system allows you to create interactive and responsive user interfaces by handling user interactions effectively. By following the JSX syntax and React's event system, you can easily manage events and build dynamic UIs in your React applications.

React Conditional Rendering

Conditional rendering in React refers to the process of rendering different content or components based on certain conditions. It allows you to control what is displayed to the user based on the current state or props of your components. There are several ways to perform conditional rendering in React:

1. Using Conditional Statements:

You can use JavaScript conditional statements like **if**, **else**, and **ternary operator** inside your JSX to conditionally render different content.

Example with ternary operator:

```
import React from 'react';

const MyComponent = ({ isLoggedIn }) => {
return (
   <div>
   {isLoggedIn ? <p>Welcome, User!</p> : <p>Please log in.</p>}
   </div>
);
};

export default MyComponent;
```

2. Using Logical && Operator:

You can use the logical AND (**&&**) operator to conditionally render content based on a condition.

Example:

```
import React from 'react';

const MyComponent = ({ isLoggedIn }) => {
return (
   <div>
   {isLoggedIn && <p>Welcome, User!</p>}
   </div>
);
};

export default MyComponent;
```

3. Using Inline If with Logical && Operator:

You can use the logical AND (**&&**) operator to conditionally render inline content based on a condition.

Example:

```
import React from 'react';

const MyComponent = ({ isLoggedIn }) => {
return (
   <div>
   {isLoggedIn && <p>Welcome, User!</p>}
   {!isLoggedIn && <p>Please log in.</p>}
   </div>
);
};

export default MyComponent;
```

4. Using Conditional Rendering with Functions:

You can define functions that return JSX elements and call them within your component's **render()** method to conditionally render content.

Example:

```
import React from 'react';

const WelcomeMessage = () => <p>Welcome, User!</p>;
const LoginMessage = () => <p>Please log in.</p>;

const MyComponent = ({ isLoggedIn }) => {
return (
   <div>
   {isLoggedIn ? <WelcomeMessage /> : <LoginMessage />}
   </div>
);
};

export default MyComponent;
```

Conditional rendering allows you to create dynamic and responsive user interfaces in your React applications. By controlling what is rendered based on specific conditions, you can provide a tailored user experience and handle different scenarios effectively. Choose the approach that best suits your use case and enhances the readability and maintainability of your code.

React Lists

In React, lists are used to render a collection of elements dynamically. Lists are commonly used when you have an array of data and you want to render a component for each item in the array. React provides a simple and efficient way to render lists using the **map()** method. Here's how you can work with lists in React:

1. Rendering Lists with map():

You can use the **map()** method to iterate over an array and generate a list of React elements based on the array items.

Example:

```
import React from 'react';

const MyList = ({ items }) => {
return (
   <ul>
   {items.map((item, index) => (
      <li key={index}>{item}</li>
   ))}
   </ul>
);
};

export default MyList;
```

2. Using Keys for List Items:

When rendering lists in React, each list item should have a unique **key** prop to help React identify which items have changed, been added, or been removed. Keys should be stable, predictable, and unique among siblings.

Example:

```
import React from 'react';

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

```
};

export default MyList;
```

3. Rendering Components in Lists:

You can render components inside lists just like any other JSX element. This allows you to create more complex UIs and reuse components within lists.

Example:

```
import React from 'react';
import ListItem from './ListItem';

const MyList = ({ items }) => {
return (
   <ul>
   {items.map((item) => (
      <ListItem key={item.id} item={item} />
   ))}
   </ul>
);
};

export default MyList;
```

4. Conditional Rendering in Lists:

You can use conditional rendering techniques within the `map()` method to conditionally render list items based on certain conditions.

Example:

```
import React from 'react';

const MyList = ({ items }) => {
return (
   <ul>
   {items.map((item) => (
      <li key={item.id}>
      {item.isVisible && <span>{item.name}</span>}
      </li>
   ))}
   </ul>
```

```
);
};

export default MyList;
```

Lists are a fundamental part of building dynamic and interactive user interfaces in React. By efficiently rendering collections of data, you can create flexible and reusable components that can adapt to different data sets and user inputs. When working with lists in React, remember to assign a unique key to each list item and leverage the power of **map()** for rendering.

React Forms

In React, forms are used to collect and handle user input. React provides a way to manage form data and handle form submission efficiently using controlled components. Controlled components keep the form data in the component's state and update it via event handlers. Here's how you can work with forms in React:

1. Basic Form Handling:

Create a form with input elements and handle user input using state and event handlers.

Example:

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

const MyForm = () => {
const [formData, setFormData] = useState({
  username: '',
  password: ''
});

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

const handleSubmit = (e) => {
  e.preventDefault();
  console.log('Form submitted:', formData);
};

return (
  <form onSubmit={handleSubmit}>
  <input
    type="text"
    name="username"
    value={formData.username}
    onChange={handleChange}
    placeholder="Username"
  />
  <input
    type="password"
    name="password"
    value={formData.password}
    onChange={handleChange}
    placeholder="Password"
  />
  <button type="submit">Submit</button>
```

```
    </form>
);
};

export default MyForm;
```

2. Form Submission:

Handle form submission by preventing the default behavior of the form submission event and performing necessary actions, such as sending data to a server or processing it locally.

Example:

```
const handleSubmit = (e) => {
    e.preventDefault();
    // Perform form submission logic here
    console.log('Form submitted:', formData);
};
```

3. Validation and Error Handling:

Implement form validation to ensure that user input meets certain criteria. Display error messages based on validation results.

Example:

```
const [errors, setErrors] = useState({});

const handleSubmit = (e) => {
e.preventDefault();
if (formData.username === '') {
    setErrors({ username: 'Username is required' });
    return;
}
// Other validation logic
console.log('Form submitted:', formData);
};

// Render error message
{errors.username && <span>{errors.username}</span>}
```

4. Handling Different Input Types:

React supports various input types such as text, password, email, checkbox, radio, select, etc. Handle different input types appropriately based on your requirements.

Example:

```
<input type="checkbox" name="subscribe" checked={formData.subscribe}
onChange={handleChange} />
```

5. Using Refs:

Use refs to interact with form elements directly, such as focusing an input field or accessing its value imperatively.

Example:

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

const MyForm = () => {
const inputRef = useRef(null);

const handleSubmit = (e) => {
  e.preventDefault();
  console.log('Input value:', inputRef.current.value);
};

return (
  <form onSubmit={handleSubmit}>
  <input type="text" ref={inputRef} />
  <button type="submit">Submit</button>
  </form>
);
};

export default MyForm;
```

React forms provide a flexible and efficient way to handle user input and create interactive user interfaces. By using controlled components, form validation, and other techniques, you can build robust and user-friendly forms in your React applications.

React Router

React Router is a popular library for handling navigation in React applications. It allows you to define routes and associate them with specific components, enabling declarative routing in your application. With React Router, you can implement features like nested routes, route parameters, and programmatic navigation.

Here's a basic example of how to use React Router in a React application:

1. **Installation**:

   First, you need to install React Router in your project. You can do this using npm or yarn:

Example:

```
npm install react-router-dom
```

or

```
yarn add react-router-dom
```

2. **Basic Setup**:

   Next, you'll set up routing in your application. Typically, you'll wrap your entire application in a **BrowserRouter** component, which provides the routing context for your application.

Example:

```
// index.js
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter as Router } from 'react-router-dom';
import App from './App';

ReactDOM.render(
<Router>
  <App />
</Router>,
document.getElementById('root')
);
```

3. **Defining Routes**:

Now, you can define routes for your application using the **Route** component. Each **Route** component maps a URL path to a specific component that should be rendered when the path matches.

Example:

```
// App.js
import React from 'react';
import { Route, Switch } from 'react-router-dom';
import Home from './Home';
import About from './About';
import NotFound from './NotFound';

const App = () => {
return (
   <div>
   <Switch>
      <Route path="/" exact component={Home} />
      <Route path="/about" component={About} />
      <Route component={NotFound} />
   </Switch>
   </div>
);
};

export default App;
```

4. Navigating Between Routes:

You can use the **Link** component provided by React Router to create navigation links. When a **Link** is clicked, React Router will navigate to the specified route without causing a full page reload.

Example:

```
// Home.js
import React from 'react';
import { Link } from 'react-router-dom';

const Home = () => {
return (
   <div>
   <h1>Home</h1>
   <Link to="/about">About</Link>
   </div>
```

```
);
};

export default Home;
```

Example:

```
// About.js
import React from 'react';
import { Link } from 'react-router-dom';

const About = () => {
return (
   <div>
   <h1>About</h1>
   <Link to="/">Home</Link>
   </div>
);
};

export default About;
```

5. Handling 404 (Not Found) Pages:

It's a good practice to have a 404 page to handle routes that don't match any defined routes. You can create a **NotFound** component and render it when no other routes match.

Example:

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

const NotFound = () => {
return <h1>404 - Not Found</h1>;
};

export default NotFound;
```

That's a basic setup of React Router in a React application. You can explore more advanced features like nested routes, route parameters, and route guards as you build more complex applications.

React Memo

React.memo is a higher-order component (HOC) provided by React that memoizes the rendering of a functional component, preventing unnecessary re-renders. It's similar to the **PureComponent** class for class components, but it's used for functional components. Memoization is a technique to optimize rendering performance by caching the result of a function call based on its input parameters.

Here's how you can use **React.memo**:

Example:

```
import React from 'react';

const MyComponent = React.memo((props) => {
/* Component logic here */
});

export default MyComponent;
```

When you wrap a functional component with **React.memo**, React will memoize the component based on its props. If the props of the component remain the same between renders, React will reuse the memoized version of the component and skip re-rendering, improving performance. However, if the props change, React will re-render the component as usual.

Here are some important points to note about **React.memo**:

1. **Shallow Comparison**:

   React.memo performs a shallow comparison of the props. It checks if the new props are shallowly equal to the previous props using **Object.is**. If the props are shallowly equal, React reuses the memoized component; otherwise, it re-renders the component.

2. **Pure Functional Components**:

   React.memo is intended for pure functional components, meaning that they don't have any side effects and their output is solely determined by their props.

3. **Custom Comparison Function**:

   You can also provide a custom comparison function as the second argument to **React.memo** to control when the component should re-render based on the props. This can be useful when you need more control over the comparison logic.

4. **Memoization Caveats**:

   Memoization is beneficial for optimizing performance in certain scenarios, but it's not always necessary or beneficial. Be cautious when using **React.memo** and ensure that it's providing a measurable performance improvement in your application.

Here's an example of using **React.memo** with a custom comparison function:

Example:

```
import React from 'react';

const MyComponent = React.memo((props) => {
/* Component logic here */
}, (prevProps, nextProps) => {
/* Custom comparison logic */
return prevProps.id === nextProps.id;
});

export default MyComponent;
```

In this example, the component will only re-render if the **id** prop changes between renders, as determined by the custom comparison function.

Styling React Using CSS

Styling React components using CSS is a common practice and can be achieved in several ways. Here are some methods to style React components using CSS:

1. **Regular CSS Stylesheets**:

   You can use regular CSS stylesheets to style your React components. Simply create `.css` files and import them into your React components. The styles defined in these CSS files will be applied to the corresponding components.

Example:

```
// Button.js
import React from 'react';
import './Button.css'; // Import CSS file

const Button = () => {
return <button className="button">Click me</button>;
};

export default Button;
```

CSS code:

```
<style>
  /* Button.css */
  .button {
    background-color: #007bff;
    color: white;
    border: none;
    padding: 10px 20px;
    border-radius: 5px;
    cursor: pointer;
  }
</style>
```

2. **CSS-in-JS Libraries**:

   CSS-in-JS libraries allow you to write CSS directly inside your JavaScript or JSX files. Popular CSS-in-JS libraries include styled-components, emotion, and CSS Modules. These libraries offer various features such as scoped styles, dynamic styles, and theming.

Example using styled-components:

```
// Button.js
import React from 'react';
import styled from 'styled-components';

const StyledButton = styled.button`
background-color: #007bff;
color: white;
border: none;
padding: 10px 20px;
border-radius: 5px;
cursor: pointer;
`;

const Button = () => {
return <StyledButton>Click me</StyledButton>;
};

export default Button;
```

3. Inline Styles:

React allows you to apply styles directly to JSX elements using inline styles. Inline styles are defined as JavaScript objects where keys represent CSS properties and values represent their respective values.

Example:

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

const Button = () => {
const buttonStyle = {
  backgroundColor: '#007bff',
  color: 'white',
  border: 'none',
  padding: '10px 20px',
  borderRadius: '5px',
  cursor: 'pointer',
};

return <button style={buttonStyle}>Click me</button>;
};

export default Button;
```

4. CSS Preprocessors:

You can use CSS preprocessors like Sass or Less to write CSS with additional features such as variables, nesting, and mixins. These preprocessors generate regular CSS files, which can then be imported into your React components.

Example with Sass:

```
<style>
  // Button.scss
  .button {
    background-color: #007bff;
    color: white;
    border: none;
    padding: 10px 20px;
    border-radius: 5px;
    cursor: pointer;
  }
</style>
```

jsxCopy code:

```
// Button.js
import React from 'react';
import './Button.scss'; // Import Sass file

const Button = () => {
return <button className="button">Click me</button>;
};

export default Button;
```

Choose the method that best fits your project requirements and preferences. Each method has its own advantages and use cases, so feel free to experiment and find the one that works best for you.

Styling React Using Sass

To style React components using Sass (Syntactically Awesome Style Sheets), you can follow these steps:

1. **Install Sass**:

   First, you need to have Sass installed in your project. You can install Sass using npm or yarn:

Example:

```
npm install node-sass
```

or

csharp code:

```
yarn add node-sass
```

2. **Create Sass Files**:

   Create `.scss` or `.sass` files to write your Sass styles. You can organize your Sass files in a way that makes sense for your project structure.

Example:

```
<style>
  // Button.scss
  .button {
    background-color: #007bff;
    color: white;
    border: none;
    padding: 10px 20px;
    border-radius: 5px;
    cursor: pointer;
  }
</style>
```

3. **Import Sass Files**:

   Import your Sass files into your React components where you want to use the styles. You can import Sass files directly into your components just like regular JavaScript or CSS files.

Example:

```
```

```
// Button.js
import React from 'react';
import './Button.scss'; // Import Sass file

const Button = () => {
return <button className="button">Click me</button>;
};

export default Button;
```

4. Use Sass Features:

Take advantage of Sass features like variables, nesting, mixins, and more to write cleaner and more maintainable styles for your React components.

Example:

```
<style>
  // variables.scss
  $primary-color: #007bff;

  // Button.scss
  @import 'variables';

  .button {
    background-color: $primary-color;
    color: white;
    border: none;
    padding: 10px 20px;
    border-radius: 5px;
    cursor: pointer;

    &:hover {
      background-color: darken($primary-color, 10%);
    }
  }
</style>
```

By following these steps, you can easily style your React components using Sass, taking advantage of its features to write more modular and maintainable stylesheets. Remember to compile your Sass files into regular CSS files using a build tool like webpack or parcel before deploying your application.

React Hooks

React Hooks are functions that enable functional components to access React features such as state and lifecycle methods. They were introduced in React 16.8 to provide a simpler and more flexible way to write components and manage stateful logic without using class components.

Here are some of the most commonly used React Hooks:

1. **useState**:

   useState allows functional components to manage state. It returns an array with the current state value and a function to update the state.

Example:

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

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

return (
   <div>
   <p>Count: {count}</p>
   <button onClick={() => setCount(count + 1)}>Increment</button>
   </div>
);
};

export default MyComponent;
```

2. **useEffect**:

   useEffect enables functional components to perform side effects, such as data fetching, subscriptions, or DOM manipulation, after every render.

Example:

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

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

useEffect(() => {
   document.title = `Clicked ${count} times`;
}, [count]);

return (
```

```
    <div>
    <p>Count: {count}</p>
    <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
);
};

export default MyComponent;
```

3. useContext:

useContext allows functional components to consume values from the React context. It accepts a context object (created with **React.createContext**) and returns the current context value.

Example:

```
import React, { useContext } from 'react';
import MyContext from './MyContext';

const MyComponent = () => {
const value = useContext(MyContext);

return <div>{value}</div>;
};

export default MyComponent;
```

4. useReducer:

useReducer is a hook that provides an alternative to **useState**. It accepts a reducer function and an initial state, returning the current state and a dispatch function to update the state.

Example:

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

const initialState = { count: 0 };

const reducer = (state, action) => {
switch (action.type) {
    case 'increment':
    return { count: state.count + 1 };
    case 'decrement':
```

```
    return { count: state.count - 1 };
    default:
    return state;
}
};

const MyComponent = () => {
const [state, dispatch] = useReducer(reducer, initialState);

return (
    <div>
    <p>Count: {state.count}</p>
    <button onClick={() => dispatch({ type: 'increment' })}>Increment</button>
    <button onClick={() => dispatch({ type: 'decrement' })}>Decrement</button>
    </div>
);
};

export default MyComponent;
```

5. useCallback and useMemo:

useCallback and **useMemo** are used to optimize performance. **useCallback** memoizes a callback function, while **useMemo** memoizes the result of a computation.

Example:

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

const MyComponent = () => {
const [count, setCount] = useState(0);
const increment = useCallback(() => setCount(count + 1), [count]);
const doubledCount = useMemo(() => count * 2, [count]);

return (
    <div>
    <p>Count: {count}</p>
    <button onClick={increment}>Increment</button>
    <p>Doubled Count: {doubledCount}</p>
    </div>
);
};

export default MyComponent;
```

React Hooks allow you to write cleaner, more concise, and easier-to-understand code compared to class components. They promote code reuse and separation of concerns by allowing you to encapsulate stateful logic and side effects into reusable functions.

React Custom Hooks

Custom Hooks in React allow you to extract and reuse stateful logic from components. They are functions that use built-in Hooks or other custom Hooks to provide a certain behavior, which can then be shared across multiple components.

Here's how you can create and use custom Hooks in React:

1. Create a Custom Hook:

You can create a custom Hook by simply defining a function that encapsulates the desired logic. Custom Hooks should start with the word "use" to comply with the Hook rules.

Example:

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

function useFetch(url) {
const [data, setData] = useState(null);
const [loading, setLoading] = useState(true);

useEffect(() => {
   async function fetchData() {
   try {
      const response = await fetch(url);
      const json = await response.json();
      setData(json);
      setLoading(false);
   } catch (error) {
      console.error('Error fetching data:', error);
   }
   }

   fetchData();
}, [url]);

return { data, loading };
}

export default useFetch;
```

2. Use the Custom Hook:

Once you have defined your custom Hook, you can use it in any functional component. Simply call the custom Hook function within the component body.

Example:

```
import React from 'react';
import useFetch from './useFetch';

const MyComponent = () => {
const { data, loading } = useFetch('https://api.example.com/data');

if (loading) {
   return <div>Loading...</div>;
}

return (
   <div>
   {/* Render data */}
   </div>
);
};

export default MyComponent;
```

3. Customize Parameters:

You can customize your custom Hook by accepting parameters. This allows you to make your Hook more flexible and reusable in different scenarios.

Example:

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

function useCounter(initialValue, step) {
const [count, setCount] = useState(initialValue);

useEffect(() => {
   const interval = setInterval(() => {
   setCount(prevCount => prevCount + step);
   }, 1000);

   return () => clearInterval(interval);
}, [step]);

return count;
}

export default useCounter;
```

4. Use the Custom Hook with Parameters:

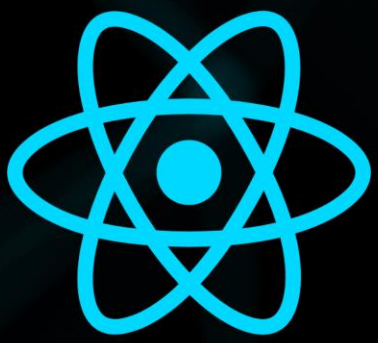
When using a custom Hook with parameters, simply pass the parameters to the Hook function when calling it.

Example:

```
import React from 'react';
import useCounter from './useCounter';

const MyComponent = () => {
const count = useCounter(0, 1);

return <div>Count: {count}</div>;
};

export default MyComponent;
```

By creating custom Hooks, you can encapsulate complex logic and share it across multiple components, promoting code reuse and maintainability in your React applications. Custom Hooks are a powerful feature of React that allows you to keep your components clean and focused on presentation logic while moving the stateful and reusable logic into separate, reusable functions.

MUHILAN ORG.



THANK YOU

+91 8838299264

muhilan6601@outlook.com