

| Title: Implementation of react concept |
| --- |

**AIM:** To demonstrate the working of React.

**Problem Definition:**

Demonstrate the use of different concept of React on the basis of following points

* Creating Environment and First project setup
* Use of JSX
* Class and function component
* Props
* States
* Event
* Event handling
* Building a basic Forms using React.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Expected OUTCOME of Experiment:

. **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Books/ Journals/ Websites referred:**

1. Express .js Deep API reference, by Azat Marden, Apress, 2nd edition, 2015.
2. <https://codeburst.io/building-a-rest-api-using-mongo-db-75cac3403fab>
3. <https://www.edureka.co/blog/rest-api-with-node-js/>
4. https://bezkoder.com/node-express-mongodb-crud-rest-api/

**Pre Lab/ Prior Concepts:**

1. Creating Environment and First Project Setup:

First, ensure you have Node.js and npm (Node Package Manager) installed on your computer.

Open your terminal and run the following command to create a new React application using Create React App:

bash

Copy code

npx create-react-app my-react-app

Navigate to the project directory:

bash

Copy code

cd my-react-app

Start the development server:

bash

Copy code

npm start

This sets up a basic React application.

2. Use of JSX:

React uses JSX (JavaScript XML) to describe the UI components. JSX allows you to write HTML-like code within your JavaScript.

jsx

Copy code

// Example JSX in a React component

function App() {

return (

<div>

<h1>Hello, React!</h1>

<p>This is a JSX example.</p>

</div>

);

}

3. Class and Function Components:

You can create React components as classes or functions. Here's an example of both:

jsx

Copy code

// Class Component

class Greeting extends React.Component {

render() {

return <h2>Hello, {this.props.name}!</h2>;

}

}

// Function Component

function Greeting(props) {

return <h2>Hello, {props.name}!</h2>;

}

4. Props:

Props (short for properties) allow you to pass data from parent to child components.

jsx

Copy code

// Using props in a component

function Welcome(props) {

return <h1>Welcome, {props.name}</h1>;

}

// Rendering the component with props

ReactDOM.render(<Welcome name="John" />, document.getElementById('root'));

5. State:

State allows you to manage data that can change over time in your component.

jsx

Copy code

// Using state in a class component

class Counter extends React.Component {

constructor() {

super();

this.state = { count: 0 };

}

render() {

return (

<div>

<p>Count: {this.state.count}</p>

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

</div>

);

}

}

6. Event Handling:

React components can handle events using event handlers.

jsx

Copy code

// Handling events in a React component

function ButtonExample() {

function handleClick() {

alert('Button clicked!');

}

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

}

7. Building a Basic Form:

You can create forms in React by using form elements and handling form submissions.

jsx

Copy code

// Building a basic form in React

class BasicForm extends React.Component {

constructor() {

super();

this.state = { inputText: '' };

}

handleChange(event) {

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

}

handleSubmit(event) {

event.preventDefault();

alert('Submitted: ' + this.state.inputText);

}

render() {

return (

<form onSubmit={(e) => this.handleSubmit(e)}>

<input

type="text"

value={this.state.inputText}

onChange={(e) => this.handleChange(e)}

/>

<button type="submit">Submit</button>

</form>

);

}

}

This example demonstrates key React concepts, from project setup to building a basic form. You can expand upon this foundation to create more complex and interactive applications.

**Implementation Details:**

# REACT COMPONENTS

1. React components are reusable HTML components that can be used everywhere.

*import* React *from* 'react'**;**

class Car *extends* React**.***Component* {

render() {

**return** <h2>Hi, I am custom component!</h2>**;**

}

}

*export* function App(**props**) {

**return** (

<div *className***=**'App'>

<h1>Hello React, This is Aatmaj.</h1>

<Car/>

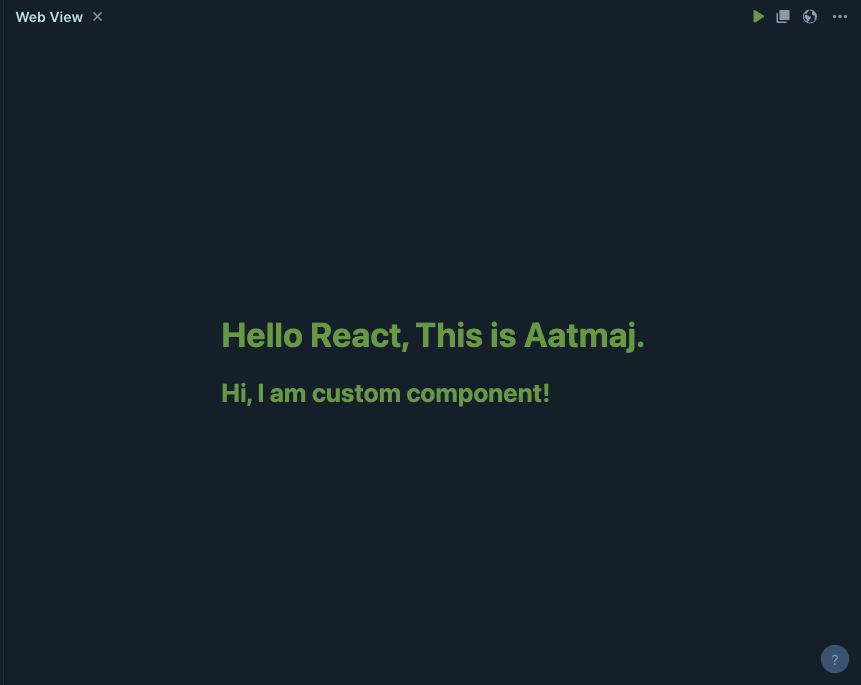
</div>

)**;**

}

*//* ***Log to console***

console**.**log('Hello console')



# FUNCTIONAL COMPONENTS

1. Functional components are reusable HTML components that can be used everywhere but with attributes as function parameters

*import* React *from* 'react'**;**

*import* ReactDOM *from* 'react-dom/client'**;**

function Car(**props**) {

**return** <h2 *style***={**{color**:**props**.***color*}**}**>I am a **{**props**.***color***}** Car!</h2>**;**

}

*export* function App(**props**) {

**return** (

<div *className***=**'App'>

<h1>Hello React, This is Aatmaj.</h1>

<Car *color***=**"red" />

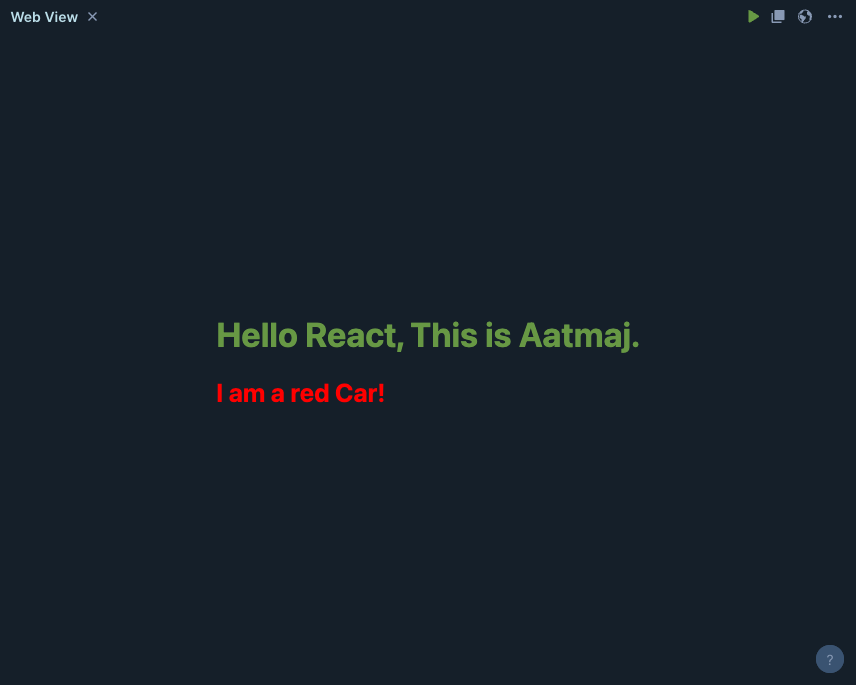
</div>

)**;**

}

*//* ***Log to console***

console**.**log('Hello console')



REACT STYLING

1. In JSX, JavaScript expressions are written inside curly braces, and since JavaScript objects also use curly braces, the styling in the example above is written inside two sets of curly braces {{}}.
2. Styling is performed using CSS and Sass

import React from 'react';

import ReactDOM from 'react-dom/client';

function Car(*props*) {

const myStyle = {

color: "red",

backgroundColor: "DodgerBlue",

padding: "10px",

fontFamily: "Sans-Serif"

};

return <h2 *style*={myStyle}>I am a {props.color} Car on a blue background!</h2>;

}

export function App(*props*) {

return (

<div *className*='App'>

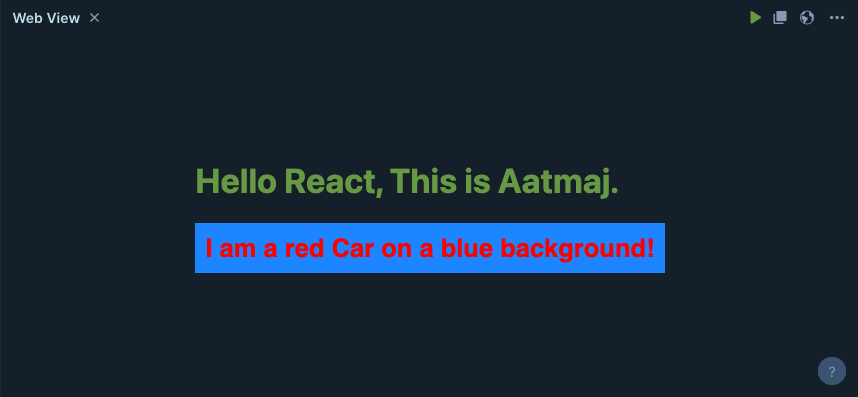
<h1>Hello React, This is Aatmaj.</h1>

<Car *color*="red" />

</div>

);

}



**Conclusion:**

**Post lab questions:**

1. **Describe the types of Component**

In React, there are two main types of components: functional components and class components. Each type has its own use cases and advantages.

1. **Functional Components:**

Also known as stateless functional components.

These are JavaScript functions that take in props as input and return JSX elements as output.

Functional components are simpler and more lightweight compared to class components.

They are primarily used for representing UI elements or "dumb" components that don't have their own internal state.

Functional components are easier to read, test, and maintain.

With the introduction of React Hooks (like useState and useEffect), functional components can also manage state and side effects, blurring the line between functional and class components.

1. **Class Components:**

Class components are ES6 classes that extend React.Component or React.PureComponent.

They have a more complex syntax compared to functional components.

Class components are used when you need to manage state or use lifecycle methods.

They are suitable for creating "smart" components that have their own internal state and handle complex logic.

Class components can be less concise and require more boilerplate code compared to functional components.

Prior to the introduction of React Hooks, class components were the primary way to manage state and side effects in React.

1. **Pure Components:**

These are similar to class components but automatically optimize rendering by performing a shallow comparison of props and state.

They re-render only if props or state change.

Typically used to optimize performance when you know that a component's render output is solely based on its props and state.

1. **Higher-Order Components (HOCs):**

These are functions that take a component and return a new component with enhanced functionality.

HOCs are used for code reuse, logic abstraction, and augmenting component behavior.

Examples include Redux's connect function for state management and libraries like react-router that provide routing capabilities.

1. **Render Props Components:**

These are components that render a function as their children, passing data and behavior to the function as arguments.

Render props are a way to share code between components and enable dynamic composition.

Commonly used for implementing features like mouse tracking, tooltips, or data fetching.