**React Introduction**

* React is a JavaScript library for building user interfaces.
* React is used to build single-page applications.
* React allows us to create reusable UI components.

**What is React?**

* React, sometimes referred to as a frontend JavaScript framework, is a JavaScript library created by Facebook.
* React is a tool for building UI components.

**How does React Work?**

* React creates a VIRTUAL DOM in memory.
* Instead of manipulating the browser's DOM directly, React creates a virtual DOM in memory, where it does all the necessary manipulating, before making the changes in the browser DOM.
* React only changes what needs to be changed!
* React finds out what changes have been made, and changes only what needs to be changed.

**React.JS History**

* Current version of React.JS is V18.0.0 (April 2022).
* Initial Release to the Public (V0.3.0) was in July 2013.
* React.JS was first used in 2011 for Facebook's Newsfeed feature.
* Facebook Software Engineer, Jordan Walke, created it.
* create-react-app includes built tools such as webpack, Babel, and ESLint.

**React Getting Started**

* To use React in production, you need npm which is included with Node.js.
* To get an overview of what React is, you can write React code directly in HTML.
* But in order to use React in production, you need npm and Node.js installed.

**React Directly in HTML**

The quickest way start learning React is to write React directly in your HTML files.

**Setting Up React Environment**

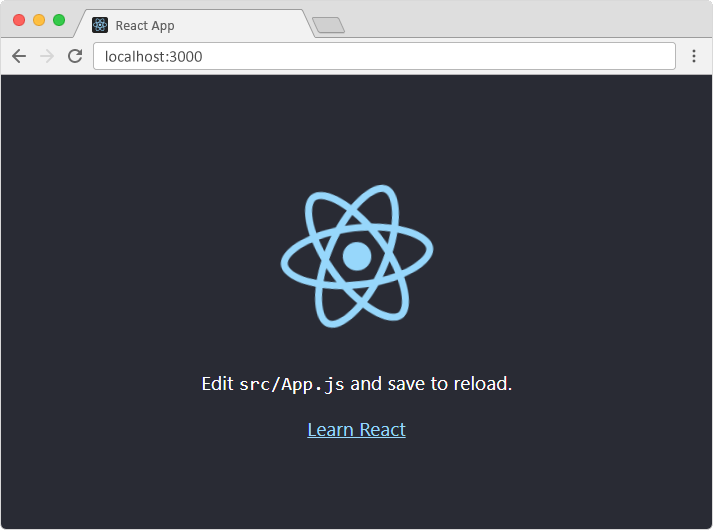
**Create react app:**

**npx create-react-app my-react-app**

**Run the React Application:**

**cd my-react-app**

**npm start**



**React directly on html:**

To use React directly in HTML, you can include React and ReactDOM from a CDN(CDN stands for **Content Delivery Network**. It is a system of distributed servers that deliver web content to users based on their geographic location, improving the speed and performance of websites by reducing latency.)and then write your components directly in a <script> tag.

1. **Including React**

**<script src="https://unpkg.com/react@17/umd/react.development.js" crossorigin></script>**

**<script> Tag**

* **Purpose**: Includes JavaScript from an external source.

**src Attribute**

* **URL**: https://unpkg.com/react@17/umd/react.development.js
  + Points to React version 17, specifically the development build.
  + **UMD**: Universal Module Definition format, compatible with various module systems.
  + **development.js**: Includes warnings and debugging features for development; use react.production.min.js for production.

**crossorigin Attribute**

* Specifies how to handle cross-origin requests, allowing safe fetching of the script from unpkg.com without credentials.

1. **Including ReactDOM**

<script src="https://unpkg.com/react-dom@17/umd/react-dom.development.js" crossorigin></script>

**URL**: https://unpkg.com/react-dom@17/umd/react-dom.development.js

* Points to ReactDOM version 17, required for rendering React components.
* Uses UMD format and is the development build.

**Purpose**: Provides methods for React to interact with the DOM, allowing components to be rendered using ReactDOM.render().

1. **Include babel**

<script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>

* **Babel**: A JavaScript compiler that enables the use of modern JavaScript features and JSX syntax, transforming them into a version compatible with older browsers.
* **Standalone Version**: Works in the browser without a build system, ideal for small projects, demos, or learning.
* **CDN**: The URL points to Babel hosted on a CDN (unpkg), allowing inclusion without local downloads.
* **babel.min.js**: Minified version for faster loading by reducing file size.

**Purpose:**

* **Transpiling JSX**: Allows the browser to understand and convert JSX syntax into standard JavaScript.

**Example Usage:**

To process JSX with Babel, use:

Html Copy code

<script type="text/babel">

// Your JSX code here

</script>

This setup lets you write React components in JSX, which Babel then converts for execution in the browser.

|  |
| --- |
| **Download react developer tool:**  <https://react.dev/learn/react-developer-tools> |

**Changes in App.js:**

**1.helloworld**

**function App() {**

**return (**

**<div className="App">**

**<h1>Hello World!</h1>**

**</div>**

**);**

**}**

**export default App;**

**2. Include a Button**

function App() {

const showAlert = () => {

alert('Button Clicked!');

};

return (

<div className="App">

<h1>Hello World!</h1>

<h2>Welcome to My First React App</h2>

<button onClick={showAlert}>Click Me!</button>

</div>

);

}

**3. Display Current Date**

**function App() {**

**const currentDate = new Date().toLocaleDateString();**

**return (**

**<div className="App">**

**<h1>Hello World!</h1>**

**<h2>Welcome to My First React App</h2>**

**<p>Today's date: {currentDate}</p>**

**</div>**

**);**

**}**

1. **Add a List of Items**

**function App() {**

**const items = ['Item 1', 'Item 2', 'Item 3'];**

**return (**

**<div className="App">**

**<h1>Hello World!</h1>**

**<h2>Welcome to My First React App</h2>**

**<ul>**

**{items.map((item, index) => (**

**<li key={index}>{item}</li>**

**))}**

**</ul>**

**</div>**

**);**

**}**

1. **Style the Component**

**function App() {**

**const appStyle = {**

**textAlign: 'center',**

**backgroundColor: '#f0f0f0',**

**padding: '20px',**

**borderRadius: '8px',**

**boxShadow: '0 4px 10px rgba(0, 0, 0, 0.1)',**

**};**

**return (**

**<div className="App" style={appStyle}>**

**<h1>Hello World!</h1>**

**<h2>Welcome to My First React App</h2>**

**<p>Today's date: {new Date().toLocaleDateString()}</p>**

**</div>**

**);**

**}**

**Changes index.js:**

**Example1**

**import React from 'react';**

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

**const myFirstElement = <h1>Hello React!</h1>**

**const root = ReactDOM.createRoot(document.getElementById('root'));**

**root.render(myFirstElement);**

**Example2**

**import React from 'react';**

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

**// Define a functional component**

**function Welcome() {**

**return <h1>Welcome to My React App!</h1>;**

**}**

**const root = ReactDOM.createRoot(document.getElementById('root'));**

**root.render(<Welcome />);**

**Example3**

**import React from 'react';**

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

**const items = ['Apple', 'Banana', 'Cherry'];**

**const myFirstElement = (**

**<ul>**

**{items.map((item, index) => (**

**<li key={index}>{item}</li>**

**))}**

**</ul>**

**);**

**const root = ReactDOM.createRoot(document.getElementById('root'));**

**root.render(myFirstElement);**

**Example4**

**import React from 'react';**

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

**const isLoggedIn = true;**

**const myFirstElement = (**

**<div>**

**<h1>{isLoggedIn ? 'Welcome back!' : 'Please log in'}</h1>**

**</div>**

**);**

**const root = ReactDOM.createRoot(document.getElementById('root'));**

**root.render(myFirstElement);**

**Example5**

**import React from 'react';**

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

**// Define a function that will be triggered on button click**

**function handleClick() {**

**alert('Button was clicked!');**

**}**

**const myFirstElement = (**

**<div>**

**<h1>Click the button below:</h1>**

**<button onClick={handleClick}>Click Me!</button>**

**</div>**

**);**

**const root = ReactDOM.createRoot(document.getElementById('root'));**

**root.render(myFirstElement);**

**React ES6**

ECMA stands for European Computer Manufacturers Association.

ECMAScript was created to standardize JavaScript, and ES6 is the 6th version of ECMAScript, it was published in 2015, and is also known as ECMAScript 2015.

**Why Should I Learn ES6?**

React uses ES6, and you should be familiar with some of the new features like:

* [Classes](https://www.w3schools.com/react/react_es6_classes.asp)
* [Arrow Functions](https://www.w3schools.com/react/react_es6_arrow.asp)
* [Variables](https://www.w3schools.com/react/react_es6_variables.asp) (let, const, var)
* [Array Methods](https://www.w3schools.com/react/react_es6_array_methods.asp) like .map()
* [Destructuring](https://www.w3schools.com/react/react_es6_destructuring.asp)
* [Modules](https://www.w3schools.com/react/react_es6_modules.asp)
* [Ternary Operator](https://www.w3schools.com/react/react_es6_ternary.asp)
* [Spread Operator](https://www.w3schools.com/react/react_es6_spread.asp)

**Classes**

A class is a type of function, but instead of using the keyword function to initiate it, we use the keyword class, and the properties are assigned inside a constructor() method.

**Example**

<!DOCTYPE html>

<html>

<body>

<script>

class Car {

constructor(name) {

this.brand = name;

}

}

const mycar = new Car("Ford");

document.write(mycar.brand);

</script>

</body>

</html>

**Method in Classes**

Example

Create a method named "present":

class Car {

constructor(name) {

this.brand = name;

}

present() {

return 'I have a ' + this.brand;

}

}

const mycar = new Car("Ford");

mycar.present();

Class Inheritance

To create a class inheritance, use the extends keyword.

A class created with a class inheritance inherits all the methods from another class.

**Example**

Create a class named "Model" which will inherit the methods from the "Car" class:

class Car {

constructor(name) {

this.brand = name;

}

present() {

return 'I have a ' + this.brand;

}

}

class Model extends Car {

constructor(name, mod) {

super(name);

this.model = mod;

}

show() {

return this.present() + ', it is a ' + this.model

}

}

const mycar = new Model("Ford", "Mustang");

mycar.show();

**React ES6 Arrow Functions**

**Arrow Functions**

Arrow functions allow us to write shorter function syntax:

**Normal function:**

hello = function() {

return "Hello World!";

}

**Arrow function:**

hello = () => {

return "Hello World!";

} or

hello = () => "Hello World!"; **Note:** This works only if the function has only one statement.

Example:

<!DOCTYPE html>

<html>

<body>

<h1>Arrow Function</h1><p id="demo"></p>

<script>

hello = (val) => "Hello " + val;

document.getElementById("demo").innerHTML = hello("World");

</script>

</body>

</html>

Example:

<h1>Arrow Function</h1>

<p>As you can see in this example, you can skip the parentheses when you have only one parameter.</p>

<p id="demo"></p>

<script>

hello = val => "Hello " + val;

document.getElementById("demo").innerHTML = hello("World");

</script>

**React ES6 Variables**

**1. let**

* Used to declare block-scoped variables (inside loops, functions, if-statements, etc.).
* Can be reassigned, but cannot be redeclared in the same scope.
* Not hoisted like var (i.e., the variable is not initialized before the declaration in the same block).

let x = 10;

x = 20; // Reassignment is allowed

**2. const**

* Used to declare block-scoped, read-only variables.
* Once assigned, a const variable cannot be reassigned.
* Useful for constants and values that should not change.

const y = 50;

// y = 60; // This would throw an error because reassignment is not allowed

**3. var**

* The old way of declaring variables (before ES6).
* Function-scoped or globally-scoped (if not inside a function).
* Can be redeclared and reassigned.
* Hoisted, meaning it is moved to the top of its scope before execution, but initialization happens where it appears in the code.

var z = 30;

z = 40; // Allowed

var z = 50; // Redeclaration is also allowed

**React ES6 Array Methods**

**Array Methods**

There are many JavaScript array methods.

One of the most useful in React is the .map() array method.

The .map() method allows you to run a function on each item in the array, returning a new array as the result.

In React, map() can be used to generate lists.

Example

Generate a list of items from an array:

[Get your server](https://www.w3schools.com/spaces/)Result Size: 246 x 575

import React from 'react';

import ReactDOM from 'react-dom/client';

const myArray = ['apple', 'banana', 'orange'];

const myList = myArray.map((item) => <p>{item}</p>)

const container = document.getElementById('root');

const root = ReactDOM.createRoot(container);

root.render(myList);

**React ES6 Destructuring**

To illustrate destructuring, we'll make a sandwich. Do you take everything out of the refrigerator to make your sandwich? No, you only take out the items you would like to use on your sandwich.

Destructuring is exactly the same. We may have an array or object that we are working with, but we only need some of the items contained in these.

Destructuring makes it easy to extract only what is needed.

Destructuring comes in handy when a function returns an array:

Example

<!DOCTYPE html>

<html>

<body>

<script>

function calculate(a, b) {

const add = a + b;

const subtract = a - b;

const multiply = a \* b;

const divide = a / b;

return [add, subtract, multiply, divide];

}

const [add, subtract, multiply, divide] = calculate(4, 7);

document.write("<p>Sum: " + add + "</p>");

document.write("<p>Difference " + subtract + "</p>");

document.write("<p>Product: " + multiply + "</p>");

document.write("<p>Quotient " + divide + "</p>");

</script>

</body>

</html>

**1. Array Destructuring**

* You can extract values from an array into individual variables.

// Basic array destructuring

const fruits = ['Apple', 'Banana', 'Cherry'];

const [first, second, third] = fruits;

console.log(first); // Output: Apple

console.log(second); // Output: Banana

console.log(third); // Output: Cherry

**2. Skipping Items in Array Destructuring**

You can skip items in the array by leaving gaps in the destructuring syntax.

const colors = ['Red', 'Green', 'Blue', 'Yellow'];

const [firstColor, , thirdColor] = colors;

console.log(firstColor); // Output: Red

console.log(thirdColor); // Output: Blue

**3. Default Values in Array Destructuring**

If the array has fewer elements than the variables, you can assign default values.

const numbers = [10];

const [a, b = 20] = numbers;

console.log(a); // Output: 10

console.log(b); // Output: 20 (default value)

**4. Object Destructuring**

You can also extract properties from objects into variables by matching their names.

const person = {

name: 'John',

age: 30,

city: 'New York'

};

const { name, age, city } = person;

console.log(name); // Output: John

console.log(age); // Output: 30

console.log(city); // Output: New York

**5. Renaming Variables in Object Destructuring**

You can rename variables during destructuring by using : newName.

const user = { username: 'alice', password: '1234' };

const { username: userName, password: pass } = user;

console.log(userName); // Output: alice

console.log(pass); // Output: 1234

const employee = {

name: 'Bob',

address: {

city: 'San Francisco',

zip: '94101'

}

};

6. **Nested Destructuring**

const { name, address: { city, zip } } = employee;

console.log(name); // Output: Bob

console.log(city); // Output: San Francisco

console.log(zip); // Output: 94101

**7. Rest or spread Operator in Destructuring**

The rest operator ... can collect the remaining elements or properties.

// Array example

const [firstItem, ...restItems] = [1, 2, 3, 4, 5];

console.log(firstItem); // Output: 1

console.log(restItems); // Output: [2, 3, 4, 5]

// Object example

const personDetails = { name: 'Anna', age: 25, country: 'USA' };

const { name, ...otherDetails } = personDetails;

console.log(name); // Output: Anna

console.log(otherDetails); // Output: { age: 25, country: 'USA' }

**React ES6 Spread Operator**

The JavaScript spread operator (...) allows us to quickly copy all or part of an existing array or object into another array or object.

<!DOCTYPE html>

<html>

<body>

<script>

const numbersOne = [1, 2, 3];

const numbersTwo = [4, 5, 6];

const numbersCombined = [...numbersOne, ...numbersTwo];

document.write(numbersCombined);

</script>

</body>

</html>

**React ES6 Modules**

**Modules**

JavaScript modules allow you to break up your code into separate files.

This makes it easier to maintain the code-base.

ES Modules rely on the import and export statements**.**

**Export**

You can export a function or variable from any file.

Let us create a file named person.js, and fill it with the things we want to export.

There are two types of exports: Named and Default.

**Named Exports**

You can create named exports two ways. In-line individually, or all at once at the bottom**.**

**Example**

**In-line individually:**

**person.js**

**export const name = "Jesse"**

**export const age = 40**

**All at once at the bottom:**

**person.js**

**const name = "Jesse"**

**const age = 40**

**export { name, age }**

**Default Exports**

Let us create another file, named message.js, and use it for demonstrating default export.

You can only have one default export in a file.

**Example**

**message.js**

const message = () => {

const name = "Jesse";

const age = 40;

return name + ' is ' + age + 'years old.';

};

export default message;

**Import**

You can import modules into a file in two ways, based on if they are named exports or default exports.

Named exports must be destructured using curly braces. Default exports do not.

Example

<!DOCTYPE html>

<html>

<body>

<p id="demo"></p>

<script type="module">

import { name, age } from "./person.js";

document.getElementById("demo").innerHTML = "My name is " + name;

document.getElementById("demo").innerHTML += ", I am " + age + ".";

</script>

</body>

</html>

**Example**

<!DOCTYPE html>

<html>

<body>

<p id="demo"></p>

<script type="module">

import message from "./message.js";

document.getElementById("demo").innerHTML = message();

</script>

</body>

</html>

**React ES6 Ternary Operator**

**Ternary Operator**

The ternary operator is a simplified conditional operator like if / else.

Syntax: condition ? expressionIfTrue : expressionIfFalse;

Here is an example using if / else:

**Normal:**

<!DOCTYPE html>

<html>

<body>

<h1 id="demo"></h1>

<script>

function renderApp() {

document.getElementById("demo").innerHTML = "Welcome!";

}

function renderLogin() {

document.getElementById("demo").innerHTML = "Please log in";

}

let authenticated = true;

if (authenticated) {

renderApp();

} else {

renderLogin();

}

</script>

<p>Try changing the "authenticated" variable to false, and run the code to see what happens.</p>

</body>

</html>

**With Ternary**

Authenticated ? renderApp() : renderLogin();

Example1

const age = 18;

const canVote = age >= 18 ? "Yes" : "No";

console.log(canVote); // Output: "Yes"

Example2(Nested Ternary Operators)

const score = 85;

const grade = score >= 90 ? 'A' : score >= 80 ? 'B' : score >= 70 ? 'C' : 'D';

console.log(grade); // Output: "B"

Example3(Using with Arrow Functions)

const isEven = (num) => num % 2 === 0 ? "Even" : "Odd";

console.log(isEven(3)); // Output: "Odd"

Example4(Conditional Rendering (in JSX))

const isLoggedIn = true;

const greeting = isLoggedIn ? <h1>Welcome back!</h1> : <h1>Please sign in.</h1>;

Example5(Assigning Values)

let userRole = 'admin';

let accessLevel = userRole === 'admin' ? 'full' : 'limited';

console.log(accessLevel); // Output: "full"

**React Render HTML**

**RENDER:**

In programming, **render** refers to the process of generating a visual representation of data or content, typically in a user interface. It transforms code (like HTML, JSX, or other templates) into a format that can be displayed on the screen, often based on the current state of the application.

Render means to display something on the screen. It takes code and turns it into visuals that users can see and interact with.

**React renders HTML to the web page by using a function called createRoot() and its method render().**

**The createRoot Function**

The createRoot() function takes one argument, an HTML element.

The purpose of the function is to define the HTML element where a React component should be displayed.

**The render Method**

The render() method is then called to define the React component that should be rendered.

But render where?

There is another folder in the root directory of your React project, named "public". In this folder, there is an index.html file.

You'll notice a single <div> in the body of this file. This is where our React application will be rendered.

INDEX.JS:

const container = document.getElementById('root');

const root = ReactDOM.createRoot(container);

root.render(<p>Hello</p>);

INDEX.HTML:

<body>

<div id="root"></div>

</body>

**The Root Node**

The root node is the HTML element where you want to display the result.

It is like a *container* for content managed by React.

It does NOT have to be a <div> element and it does NOT have to have the id='root':

<body>

<header id="sandy"></header>

</body>

const container = document.getElementById('sandy');

const root = ReactDOM.createRoot(container);

root.render(<p>Hallo</p>);

**React JSX**

**What is JSX?**

JSX stands for JavaScript XML.

JSX allows us to write HTML in React.

JSX makes it easier to write and add HTML in React.

**Coding JSX**

JSX allows us to write HTML elements in JavaScript and place them in the DOM without any createElement()  and/or appendChild() methods.

JSX converts HTML tags into react elements.

**Example 1**

**JSX:**

const myElement = <h1>I Love JSX!</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myElement);

Example 2

Without JSX:

const myElement = React.createElement('h1', {}, 'I do not use JSX!');

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myElement);

**Expressions in JSX**

With JSX you can write expressions inside curly braces { }.

The expression can be a React variable, or property, or any other valid JavaScript expression. JSX will execute the expression and return the result:

import React from 'react';

import ReactDOM from 'react-dom/client';

const myElement = <h1>React is {5 + 5} times better with JSX</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myElement);

**Inserting a Large Block of HTML**

To write HTML on multiple lines, put the HTML inside parentheses:

import React from 'react';

import ReactDOM from 'react-dom/client';

const myElement = (

<ul>

<li>Apples</li>

<li>Bananas</li>

<li>Cherries</li>

</ul>

);

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myElement);

**One Top Level Element**

The HTML code must be wrapped in *ONE* top level element.

So if you like to write *two* paragraphs, you must put them inside a parent element, like a div element.

import React from 'react';

import ReactDOM from 'react-do/client';

const myElement = (

<div>

<h1>I am a Header.</h1>

<h1>I am a Header too.</h1>

</div>

);

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myElement);

**NOTE:** JSX will throw an error if the HTML is not correct, or if the HTML misses a parent element.

Alternatively, you can use a "fragment" to wrap multiple lines. This will prevent unnecessarily adding extra nodes to the DOM.

A fragment looks like an empty HTML tag: <></>.

import React from 'react';

import ReactDOM from 'react-dom/client';

const myElement = (

<>

<p>I am a paragraph.</p>

<p>I am a paragraph too.</p>

</>

);

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myElement);

**Elements Must be Closed**

JSX follows XML rules, and therefore HTML elements must be properly closed.

**NOTE:** JSX will throw an error if the HTML is not properly closed.

import React from 'react';

import ReactDOM from 'react-dom/client';

const myElement = <input type="text" />;

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myElement);

**Attribute class = className**

The class attribute is a much used attribute in HTML, but since JSX is rendered as JavaScript, and the class keyword is a reserved word in JavaScript, you are not allowed to use it in JSX.

**NOTE:** Use attribute className instead.

JSX solved this by using className instead. When JSX is rendered, it translates className attributes into class attributes.

import React from 'react';

import ReactDOM from 'react-dom/client';

const myElement = <h1 className="myclass">Hello World</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myElement);

**Conditions - if statements**

React supports if statements, but not *inside*JSX.

To be able to use conditional statements in JSX, you should put the if statements outside of the JSX, or you could use a ternary expression instead:

**Option 1:**

Write if statements outside of the JSX code:

import React from 'react';

import ReactDOM from 'react-dom/client';

const x = 5;

let text = "Goodbye";

if (x < 10) {

text = "Hello";

}

const myElement = <h1>{text}</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myElement);

**Option 2:**

Use ternary expressions instead:

import React from 'react';

import ReactDOM from 'react-dom/client';

const x = 5;

const myElement = <h1>{(x) < 10 ? "Hello" : "Goodbye"}</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myElement);

**React Components**

Components are like functions that return HTML elements.

Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and return HTML.

Components come in two types, Class components and Function components, in this tutorial we will concentrate on Function components.

**Work in isolation:**

React components work in isolation to promote:

1. **Encapsulation**: Each component manages its own logic and state.
2. **Reusability**: Components can be reused across the application.
3. **Maintainability**: Easier to update without affecting others.
4. **Testing**: Simplified unit testing for individual components.
5. **Separation of Concerns**: Clear distinction between UI and logic.
6. **Props and State Management**: Unidirectional data flow keeps data handling straightforward.

This modular approach leads to cleaner, more manageable code.

**Create Your First Component**

When creating a React component, the component's name *MUST* start with an uppercase letter.

**Class Component**

A class component must include the extends React.Component statement. This statement creates an inheritance to React.Component, and gives your component access to React.Component's functions.

The component also requires a render() method, this method returns HTML.

**Example1**

**Car.js** **(the component file)**

class Car extends React.Component {

render() {

return <h2>Hi, I am a Car!</h2>;

}

}

Usage: App.js (the main application file)

import React from 'react';

import ReactDOM from 'react-dom';

import Car from './Car'; // Import the Car component

const App = () => {

return (

<div>

<h1>Welcome to My Garage!</h1>

<Car /> {/\* Render the Car component \*/}

</div>

);

};

ReactDOM.render(<App />, document.getElementById('root')); // Render App into the DOM

**Example2**

**Simple Message Component**

**SimpleMessage.js (the component file)**

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

**class SimpleMessage extends Component {**

**render() {**

**return <h1>This is a simple message!</h1>;**

**}**

**}**

**export default SimpleMessage;**

Usage: **App.js** (the main application file)

import React from 'react';

import ReactDOM from 'react-dom';

import SimpleMessage from './SimpleMessage'; // Import the SimpleMessage component

const App = () => {

return (

<div>

<SimpleMessage /> {/\* Render the SimpleMessage component \*/}

</div>

);

};

ReactDOM.render(<App />, document.getElementById('root')); // Render App into the DOM

**Example3**

**Counter Component.js**

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

**class Counter extends Component {**

**constructor(props) {**

**super(props);**

**this.state = { count: 0 };**

**}**

**increment = () => {**

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

**};**

**render() {**

**return (**

**<div>**

**<h1>Count: {this.state.count}</h1>**

**<button onClick={this.increment}>Increment</button>**

**</div>**

**);**

**}**

**}**

**export default Counter;**

**Usage: App.js (the main application file)**

**import React from 'react';**

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

**import Counter from './Counter'; // Import the Counter component**

**const App = () => {**

**return (**

**<div>**

**<h1>Welcome to the Counter App!</h1>**

**<Counter /> {/\* Render the Counter component \*/}**

**</div>**

**);**

**};**

**ReactDOM.render(<App />, document.getElementById('root')); // Render App into the DOM**

**Example4**

**Greeting Component.js**

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

**class Greeting extends Component {**

**render() {**

**return <h1>Hello, {this.props.name}!</h1>;**

**}**

**}**

**export default Greeting;**

**Usage: App.js (the main application file)**

**import React from 'react';**

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

**import Greeting from './Greeting'; // Import the Greeting component**

**const App = () => {**

**return (**

**<div>**

**<h1>Welcome to My App!</h1>**

**<Greeting name="Alice" /> {/\* Pass a name as a prop \*/}**

**</div>**

**);**

**};**

**ReactDOM.render(<App />, document.getElementById('root')); // Render App into the DOM**

**Example5**

**Toggle Visibility Component.js**

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

**class ToggleMessage extends Component {**

**constructor(props) {**

**super(props);**

**this.state = { isVisible: false };**

**}**

**toggleVisibility = () => {**

**this.setState({ isVisible: !this.state.isVisible });**

**};**

**render() {**

**return (**

**<div>**

**<button onClick={this.toggleVisibility}>**

**{this.state.isVisible ? 'Hide' : 'Show'} Message**

**</button>**

**{this.state.isVisible && <p>This is a toggled message!</p>}**

**</div>**

**);**

**}**

**}**

**export default ToggleMessage;**

**Usage:App.js** (the main application file)

import React from 'react';

import ReactDOM from 'react-dom';

import ToggleMessage from './ToggleMessage'; // Import the ToggleMessage component

const App = () => {

return (

<div>

<h1>Welcome to the Toggle Message App!</h1>

<ToggleMessage /> {/\* Render the ToggleMessage component \*/}

</div>

);

};

ReactDOM.render(<App />, document.getElementById('root')); // Render App into the DOM

**REACT HOOK**

**USESTATE:**