# Introduction to React

## What is React and Why Use It?

React is a popular JavaScript library developed by Facebook (now Meta) for building user interfaces (UIs), particularly for single-page applications (SPAs). It allows developers to create complex UIs from small, reusable components.

### Key Features of React:

1. **Component-based Architecture**: React applications are built using small, reusable components. These components manage their own state and can be combined to form larger UIs.
2. **Virtual DOM**: React uses a virtual DOM to improve performance. Instead of re-rendering the entire UI when changes occur, React updates only the parts of the UI that have changed, leading to faster rendering and improved user experience.
3. **Declarative Syntax**: React allows developers to describe what the UI should look like based on the state, and React automatically updates the UI whenever the state changes.
4. **One-way Data Binding**: React follows unidirectional data flow, which means that data flows in one direction: from parent to child components. This makes the flow of data easy to track and debug.

## Setting Up a React Project Using Create-React-App

create-react-app is a command-line tool that sets up a React project with a good default configuration, allowing you to skip the initial setup and focus on writing your code.

### Steps to Set Up a React Project:

1. **Install Node.js and npm**:

React requires Node.js and npm (Node package manager) to run.

You can download Node.js from [here](https://nodejs.org/), and npm comes bundled with it.

1. **Install create-react-app**: Open a terminal and run:

npx create-react-app my-first-app

This will create a folder called my-first-app with all the necessary files and configurations for a React project.

1. **Navigate into the project folder**:

cd my-first-app

1. **Start the React development server**: Run the following command to start a local development server and see your React app:

npm start

This will open a browser window with your app running at http://localhost:3000.

Indeed, **create-react-app** is deprecated, and many developers are now opting for **Vite** due to its superior build speed and modern features.

Here's a step-by-step guide to create a React project using **Vite**:

Vite is a modern build tool that works faster than traditional bundlers like Webpack. You don't need to install anything globally to start using Vite.

You can directly create a new project using the npm or yarn commands.

#### Option 1: Using npm (Recommended)

1. In your terminal, run the following command to create a new React project using Vite:

npm create vite@latest my-react-app --template react

* + **vite@latest** specifies the version of Vite to use.
  + **my-react-app** is the name of the new React project. You can replace this with any name you prefer.
  + **--template react** specifies that you want to use the React template.

1. After running the command, you will be prompted with options for additional configuration. For example:

Where would you like to create your app?

Confirm or customize the project location.

#### Option 2: Using yarn (If you're using yarn as a package manager)

1. You can use yarn as well if it’s your preferred package manager. Run:

yarn create vite@latest my-react-app --template react

cd my-react-app

Now, install the required dependencies for your new project using npm (or yarn).

* If using **npm**:

npm install

* If using **yarn**:

Yarn install

This will install all the necessary dependencies, including React, ReactDOM, and Vite-specific configurations.

Once the dependencies are installed, you can start the Vite development server.

* If using **npm**:

npm run dev

* If using **yarn**:

yarn dev

Vite will start the development server, and you should see an output like:

VITE vX.X.X ready in Yms

Local: http://localhost:3000/

Network: use --host to expose

* Open your browser and go to [http://localhost:3000](http://localhost:3000/). You should see your new React app running!

After creating the project, you will have a structure similar to this:

my-react-app/

├── index.html # Entry point HTML file

├── src/ # Source code directory

│ ├── App.jsx # Main App component

│ ├── main.jsx # Main entry point for React (JSX)

│ └── assets/ # For static files like images, fonts, etc.

├── package.json # Project metadata and dependencies

├── vite.config.js # Vite configuration file

└── node\_modules/ # Installed dependencies

### ****Modify and Start Developing****

* **App.jsx** is the entry point for your React app, and you can start modifying it to build your React UI.
* **vite.config.js** contains Vite’s configuration, but you typically won’t need to modify this unless you need custom configurations.

### ****Building for Production****

Once you're ready to deploy your app, Vite makes it easy to create a production build:

* If using **npm**:

npm run build

* If using **yarn**:

yarn build

This will create a dist/ folder that contains the optimized, minified production build of your app, ready for deployment.

# JSX (JavaScript XML)

## What is JSX?

JSX (JavaScript XML) is a syntax extension for JavaScript that looks similar to HTML but is used with React to describe what the UI should look like. It allows developers to write HTML-like code inside JavaScript, which React will then transform into the actual HTML elements that the browser can render.

### JSX in React:

React transforms JSX into JavaScript function calls (using React.createElement). These function calls are then used to generate actual HTML in the DOM. JSX is not mandatory, but it is highly recommended because it simplifies the code and enhances readability.

### Example 1: Functional Component with JSX

Here’s an example of how JSX is used inside a functional component:

import React from 'react';

function WelcomeMessage() {

return (

<div>

<h1>Welcome to React!</h1>

<p>Learning React is fun and rewarding.</p>

</div>

);

}

export default WelcomeMessage;

### Explanation:

* In this example, the WelcomeMessage component returns JSX.
* The <div>, <h1>, and <p> tags look like HTML, but they are actually JSX syntax.
* When React renders this component, it will convert the JSX into JavaScript function calls.

### Example 2: Class Component with JSX

For class components, JSX is typically used inside the render method:

import React, { Component } from 'react';

class WelcomeMessage extends Component {

render() {

return (

<div>

<h1>Welcome to React!</h1>

<p>Learning React is fun and rewarding.</p>

</div>

);

}

}

export default WelcomeMessage;

## JSX Expressions: Embedding JavaScript in JSX

JSX allows you to embed JavaScript expressions within the markup by wrapping them in curly braces {}. These expressions are evaluated as JavaScript code, and their result is rendered inside the JSX.

You can embed variables directly in JSX by wrapping them in curly braces:

JSX allows you to use JavaScript expressions, such as math or conditional logic, inside curly braces:

import React from 'react';

function Sum() {

const a = 5;

const b = 10;

return (

<div>

<p>The sum of {a} and {b} is {a + b}.</p>

</div>

);

}

export default Sum;

In this case, {a + b} will be evaluated, and the result, 15, will be displayed in the rendered output.

### Conditional Rendering in JSX

You can also use conditional expressions inside JSX to render content dynamically:

import React from 'react';

function Greeting({ isLoggedIn }) {

return (

<div>

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

</div>

);

}

export default Greeting;

Here, the ternary operator is used to conditionally render either "Welcome back!" or "Please log in." based on the value of isLoggedIn.

## JSX and HTML Differences

Although JSX looks very similar to HTML, there are several important differences to be aware of when using it in React:

### 1. ****Class vs className****

In HTML, you use the class attribute to assign CSS classes to an element. In JSX, however, you use className instead because class is a reserved word in JavaScript.

<div className="container">

<h1>Hello, World!</h1>

</div>

### 2. ****for vs htmlFor****

In HTML, the for attribute is used to associate a label with an input element. In JSX, for is a reserved word, so React uses htmlFor instead.

<label htmlFor="username">Username</label>

<input type="text" id="username" />

### 3. ****Self-closing Tags****

In JSX, self-closing tags must be explicitly closed with a /, even for tags like <img>, <input>, or <br>. In HTML, these tags are typically written without a closing slash.

<img src="image.jpg" alt="description" />

<input type="text" />

<br />

### 4. ****Style Attribute****

In JSX, inline styles are written as an object, where the property names use camelCase instead of hyphenated lowercase (e.g., backgroundColor instead of background-color).

<div style={{ backgroundColor: 'blue', color: 'white' }}>

Styled Content

</div>

### 5. ****JavaScript Expressions****

JSX allows you to embed JavaScript expressions directly inside curly braces {}. HTML doesn't have a built-in way to do this.

<p>The current time is {new Date().toLocaleTimeString()}</p>

### 6. ****Boolean Attributes****

In HTML, attributes like checked, disabled, or readonly are boolean attributes, meaning they are either present or absent. In JSX, you must assign a value (true or false) to these attributes.

<input type="checkbox" checked={true} />

<button disabled={false}>Click Me</button>

## Functional Components with State (Using useState Hook):

### Syntax of useState:

const [state, setState] = useState(initialState);

* state: The current state value.
* setState: A function that updates the state.
* initialState: The initial value of the state.

### Example 1: Counter Component Using useState

import React, { useState } from 'react';

function Counter() {

// Declare state variable 'count' with initial value 0

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

// Function to increment count

const increment = () => setCount(count + 1);

// Function to decrement count

const decrement = () => setCount(count - 1);

return (

<div>

<h1>Count: {count}</h1>

<button onClick={increment}>Increment</button>

<button onClick={decrement}>Decrement</button>

</div>

);

}

export default Counter;

### Explanation:

* const [count, setCount] = useState(0);: Here, count is the state variable, and setCount is the function to update it. The initial value of count is set to 0.
* increment and decrement functions use setCount to update the state by adding or subtracting 1.

### Updating State:

When setCount(count + 1) is called, React will update the count value and re-render the component to reflect the new value.

### Updating State with Objects

If the state is an object, you can update a property of the object by spreading the previous state and modifying only the part that changes.

import React, { useState } from 'react';

function UserProfile() {

const [user, setUser] = useState({

name: 'John Doe',

age: 30

});

const updateName = () => {

setUser((prevUser) => ({

...prevUser, // Spread the previous state

name: 'Jane Smith' // Update the name

}));

};

return (

<div>

<h1>{user.name}, Age: {user.age}</h1>

<button onClick={updateName}>Change Name</button>

</div>

);

}

export default UserProfile;

### Explanation:

* setUser((prevUser) => ({ ...prevUser, name: 'Jane Smith' })): This spreads the previous user object and updates only the name property.
* The updateName function changes the name in the state, and React re-renders the component to reflect the updated state.

## Class Components with State

A **Class Component** is a more traditional way of creating components in React, and it was used prior to the introduction of React Hooks. Class components are JavaScript classes that extend React.Component and must have a render method that returns JSX.

Before **React Hooks**, state was only available in **class components**. To manage state in class components, we used this.state and this.setState.

### Class Component State Syntax:

class MyComponent extends React.Component {

constructor(props) {

super(props);

this.state = {

// Initial state

};

}

render() {

return (

<div>

{/\* Use state here \*/}

</div>

);

}

}

### Example: Counter in Class Component

import React, { Component } from 'react';

class Counter extends Component {

constructor(props) {

super(props);

// Initial state

this.state = {

count: 0

};

}

// Function to increment count

increment = () => {

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

};

// Function to decrement count

decrement = () => {

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

};

render() {

return (

<div>

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

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

<button onClick={this.decrement}>Decrement</button>

</div>

);

}

}

export default Counter;

### Explanation:

* this.state: The state object is initialized inside the constructor with this.state = { count: 0 }.
* this.setState: This method is used to update the state. When this.setState is called, React schedules a re-render of the component.

## Key Differences Between State in Functional and Class Components

1. **State Management**:
   * In **functional components**, we use the useState hook to manage state.
   * In **class components**, we use this.state for state initialization and this.setState to update state.
2. **State Update**:
   * In **functional components**, state updates are done by calling the state setter function returned by useState.
   * In **class components**, state updates are done using this.setState.
3. **Re-rendering**:
   * In **both functional and class components**, when the state changes, React automatically re-renders the component to reflect the updated state.

## Component Properties (Props)

**Props (short for properties)** are a way to pass data from one component to another in React. They are read-only and are passed from parent components to child components.

### How Props Work:

Props are passed to components as attributes in JSX and can be accessed inside the component via props.

### Example of Passing Props to Functional Components:

import React from 'react';

function Greeting({ name }) {

return <h1>Hello, {name}!</h1>;

}

function App() {

return <Greeting name="Alice" />;

}

export default App;

In this example:

* Greeting is a functional component that accepts a name prop.
* The App component passes "Alice" as a value for the name prop when rendering the Greeting component.

### Example of Passing Props to Class Components:

import React, { Component } from 'react';

class Greeting extends Component {

render() {

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

}

}

class App extends Component {

render() {

return <Greeting name="Alice" />;

}

}

export default App;

In this example:

* The Greeting class component accesses the name prop using this.props.name.
* The App class component passes "Alice" as a value for the name prop.

## 4. Rendering Components in React

Rendering components in React refers to displaying the component's UI on the screen. Components can be rendered by simply calling them in JSX.

### Rendering a Component:

You can render a component by simply using it as an HTML tag inside another component’s JSX.

import React from 'react';

import ReactDOM from 'react-dom';

function App() {

return (

<div>

<h1>Hello, React!</h1>

<p>This is my first React app.</p>

</div>

);

}

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

In this example:

* ReactDOM.render is used to render the App component into the HTML element with the id of root.
* The App component contains JSX, which is rendered by React.

## 5. Passing Props to Child Components

You can pass data from a parent component to a child component using props. This allows you to create dynamic and reusable components.

### Example 1: Passing Props to Child Component (Functional Components)

import React from 'react';

function Parent() {

const message = "Hello from Parent!";

return <Child message={message} />;

}

function Child({ message }) {

return <h2>{message}</h2>;

}

export default Parent;

In this example:

* The Parent component passes the message prop to the Child component.
* The Child component receives this prop and renders it inside an <h2> tag.

### Example 2: Passing Props to Child Component (Class Components)

import React, { Component } from 'react';

class Parent extends Component {

render() {

const message = "Hello from Parent!";

return <Child message={message} />;

}

}

class Child extends Component {

render() {

return <h2>{this.props.message}</h2>;

}

}

export default Parent;

Here:

* The Parent class component passes the message prop to the Child class component.
* The Child class component accesses the message prop via this.props.message and renders it.

## Example Project: Building a User Profile Card (Using Props)

Let’s combine all the concepts to create a simple **User Profile Card** that passes user information (like name, age, and bio) from a parent component to a child component.

### Step-by-Step Code Creation:

1. **Create the React App**: If you haven’t already done so, create a new React app:

npx create-react-app user-profile

cd user-profile

1. **Create the Profile Component**: Inside src/App.js, define a ProfileCard component to display user information:

import React from 'react';

function ProfileCard({ name, age, bio }) {

return (

<div style={{ border: '1px solid #ddd', padding: '20px', width: '300px', borderRadius: '10px' }}>

<h2>{name}</h2>

<p>Age: {age}</p>

<p>{bio}</p>

</div>

);

}

function App() {

const user = {

name: 'John Doe',

age: 30,

bio: 'Software developer with a passion for learning and building amazing things.'

};

return <ProfileCard name={user.name} age={user.age} bio={user.bio} />;

}

export default App;

1. **Run the Application**: Save the changes and run the app using:

npm start

# Event Handling in React

In this chapter, we’ll explore **event handling** in React, which allows you to make your app interactive by responding to user actions like clicks, form submissions, key presses, and more. Event handling in React is very similar to how it’s done in plain JavaScript, but with some important React-specific differences.

## 1. Handling Events in React

In React, event handling is slightly different from traditional HTML event handling. Instead of using the standard HTML event names (like onclick or onsubmit), React uses camelCase for event names (like onClick or onSubmit). React also passes a **synthetic event** to the handler, which is a cross-browser wrapper around the native JavaScript event.

### Syntax of Event Handling in React:

<button onClick={this.handleClick}>Click me</button>

Here, onClick is the event handler that listens for clicks on the button, and handleClick is the method that gets called when the button is clicked.

### Example: Handling Click Events in React

import React from 'react';

function ClickButton() {

const handleClick = () => {

alert('Button clicked!');

};

return (

<div>

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

</div>

);

}

export default ClickButton;

In this example:

* The handleClick function is called when the button is clicked.
* The function displays an alert saying "Button clicked!".

### Common Event Types in React:

* onClick: Triggered when a mouse click occurs.
* onChange: Triggered when the value of a form element (like an input field) changes.
* onSubmit: Triggered when a form is submitted.
* onKeyPress: Triggered when a key is pressed.
* onMouseEnter/onMouseLeave: Triggered when the mouse enters or leaves an element.

### Binding with Arrow Functions:

Another approach is to use an **arrow function** in the event handler, which automatically binds this to the current instance.

#### Example of Binding with Arrow Functions:

import React, { Component } from 'react';

class ClickButton extends Component {

handleClick = () => {

alert('Button clicked!');

};

render() {

return (

<div>

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

</div>

);

}

}

export default ClickButton;

In this example:

* The handleClick method is defined as an arrow function, which automatically binds this to the class instance.
* When the button is clicked, this.handleClick() is executed, and this correctly refers to the instance of the class.

## 3. Passing Arguments to Event Handlers

Sometimes, you may need to pass additional arguments to your event handler.

#### Example: Passing Arguments with Arrow Functions

import React from 'react';

function ClickButton() {

const handleClick = (name) => {

alert(`Button clicked by ${name}!`);

};

return (

<div>

<button onClick={() => handleClick('Alice')}>Click Me!</button>

</div>

);

}

export default ClickButton;

In this example:

* The handleClick function accepts a name argument.
* When the button is clicked, the handleClick function is called with the argument 'Alice'.

# Conditional Rendering in React

In this chapter, we'll learn about **conditional rendering** in React, which allows us to display different content based on certain conditions. This is essential for making your app dynamic and responsive to user input or changes in state.

## 1. Rendering Based on Conditions

In React, you can render different content depending on a condition. This is commonly done with if statements or the **ternary operator**.

### Using if Statements for Conditional Rendering

You can use a regular if statement to conditionally render content inside your component’s render method or the function body of a functional component. However, remember that you cannot use if directly inside JSX. You need to return JSX from a function based on the condition.

#### Example: Using if for Conditional Rendering in a Functional Component

import React, { useState } from 'react';

function LoginStatus() {

const [isLoggedIn, setIsLoggedIn] = useState(false);

if (isLoggedIn) {

return <h1>Welcome Back!</h1>;

} else {

return <h1>Please Log In</h1>;

}

}

export default LoginStatus;

### Explanation:

* We use an if statement to check the value of isLoggedIn. If it's true, the component renders "Welcome Back!". If it's false, it renders "Please Log In".
* The condition can depend on user actions, data fetching, or other dynamic aspects of the app.

### Using Ternary Operators for Conditional Rendering

The **ternary operator** is a concise way to write if-else statements and is often used directly inside JSX. The syntax is:

condition ? expr1 : expr2;

Where expr1 is the result when the condition is true, and expr2 is the result when the condition is false.

#### Example: Using Ternary Operator for Conditional Rendering

import React, { useState } from 'react';

function LoginStatus() {

const [isLoggedIn, setIsLoggedIn] = useState(false);

return (

<div>

{isLoggedIn ? <h1>Welcome Back!</h1> : <h1>Please Log In</h1>}

</div>

);

}

export default LoginStatus;

### Explanation:

* The isLoggedIn condition is checked using a ternary operator. If isLoggedIn is true, it shows "Welcome Back!", otherwise it shows "Please Log In".
* The ternary operator is often preferred in JSX because of its brevity and readability.

## 2. Rendering Lists Using .map()

In React, it's common to render lists of items. You can use the .map() function to iterate over an array and return an array of JSX elements.

### Basic Syntax of .map():

array.map((item, index) => {

return <li key={index}>{item}</li>;

});

* item is the current element of the array.
* index is the index of the current item in the array.
* The key prop is essential in lists for React to track each element’s identity when the list changes.

### Example: Rendering a List of Items with .map()

import React from 'react';

function ItemList() {

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

return (

<div>

<h1>Fruits</h1>

<ul>

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

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

))}

</ul>

</div>

);

}

export default ItemList;

### Explanation:

* items is an array containing the names of fruits.
* .map() iterates over the array, creating a <li> element for each item in the list.
* Each <li> element is given a unique key prop to help React optimize rendering.

### Using Conditional Rendering in Lists

You can also combine conditional rendering with .map() to render different content based on conditions.

#### Example: Filtering Items and Rendering Conditionally

import React from 'react';

function ItemList() {

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

const filterCondition = 'A'; // Only show items that start with "A"

return (

<div>

<h1>Fruits</h1>

<ul>

{items

.filter(item => item.startsWith(filterCondition))

.map((item, index) => (

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

))}

</ul>

</div>

);

}

export default ItemList;

### Explanation:

* The .filter() method filters the items array to only include items that start with the letter "A".
* The .map() function then renders each filtered item as a list element.

### Rendering Lists of Objects

If your list contains objects, you can render multiple properties from the object. Make sure to use the key prop properly to avoid React warnings.

#### Example: Rendering a List of Objects

import React from 'react';

function PersonList() {

const people = [

{ id: 1, name: 'Alice', age: 25 },

{ id: 2, name: 'Bob', age: 30 },

{ id: 3, name: 'Charlie', age: 35 }

];

return (

<div>

<h1>People</h1>

<ul>

{people.map((person) => (

<li key={person.id}>

{person.name} - {person.age} years old

</li>

))}

</ul>

</div>

);

}

export default PersonList;

### Explanation:

* The people array contains objects with properties id, name, and age.
* We use .map() to render each person's name and age, and we use the id as the unique key prop.