ReactJS is a popular open-source JavaScript library for building user interfaces, primarily single-page applications. It enables developers to create reusable UI components and efficiently update and render them in response to changing data. React is maintained by Meta (formerly Facebook) and a community of developers.

Key Features:

- **Component-Based Architecture**: Build encapsulated components that manage their state.
- **Declarative:** React makes it easy to design interactive UIs.
- **Virtual DOM:** Improves performance by minimizing direct DOM manipulation.
- **One-Way Data Flow:** Data flows from parent to child components, making it predictable.

Installing Node.js Server

Before we can create a React application, we need to have Node.js installed on our system. Node.js provides the runtime environment for running JavaScript code outside the browser and includes npm (Node Package Manager), which is essential for managing dependencies.

Download and Install Node.js:

- ✓ Go to the Node.js official website.
- ✓ Download the LTS version (recommended for most users).
- ✓ Install Node.js using the installer for your operating system.

Verify Installation:

- Open a terminal or command prompt.
- Type node -v to check the installed Node.js version.
- Type npm -v to check the installed npm version.

Creating a Simple React Project

React projects are typically created using the create-react-app command, a pre-configured tool that sets up the project structure and configuration.

Install create-react-app (Optional Step):

Globally install it with:

npm install -g create-react-app

Create a New Project:

Run the following command in your terminal:

- npx **create-react-app** my-app
- Replace my-app with your desired project name.

Navigate to our Project Directory:

cd my-app

Start the Development Server:

npm start

This will open the React application in your default web browser at http://localhost:3000.

Explore the Project Structure:

src/: Contains the main application files.
public/: Holds static assets.
package.json: Includes project metadata and dependencies.

JSX (JavaScript XML) is a syntax extension for JavaScript commonly used with React to describe the structure of user interfaces. It allows you to write HTML-like code directly in JavaScript, making it easier to visualize the UI while maintaining the flexibility of JavaScript.

Key Concepts of JSX in Templating:

Embedding Expressions:

JSX allows embedding JavaScript expressions inside curly braces {} within the HTML-like syntax. This enables dynamic rendering of values or invoking functions to display UI elements.

```
const userName = "John";
return <h1>Hello, {userName}!</h1>;
```

Conditional Rendering:

JSX supports conditional rendering using JavaScript constructs like if statements, ternary operators, or logical operators.

```
const isLoggedIn = true;
return (
  <div>
    {isLoggedIn ? <p>Welcome back!</p> : <p>Please log in.</p>}
  </div>
);
```

Attributes in JSX:

Attributes in JSX are written similarly to HTML, but with camelCase for property names (e.g., className instead of class, onClick instead of onclick).

```
const buttonText = "Click me";
return <button onClick={handleClick}>{buttonText}</button>;
```

JSX Elements Are Expressions:

We can assign JSX to variables and return them from functions. Since JSX is just JavaScript, it can be used anywhere you'd use a regular function or object.

```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {user}!</h1>;
  }
  return <h1>Hello, stranger.</h1>;
}
```

Fragments:

In JSX, we can use fragments (<>...</>) to group elements without adding extra nodes to the DOM.

```
return (
  <>
    <h1>Heading</h1>
    <p>This is a paragraph.</p>
```

```
  </>
);
```

Components and Reusability:
JSX works seamlessly with React components, allowing for reusable, dynamic templates. You can create custom components and reuse them across your application.

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

return <Welcome name="Sara" />;
```

Advantages of Using JSX for Templating:
Readability: JSX closely resembles HTML, making it easier for developers to understand the structure of the UI.
Dynamic UI: Integration with JavaScript allows for dynamic and interactive UI elements.
Component-based structure: Encourages the creation of reusable components, enhancing maintainability and scalability.

In React, components are the building blocks of any React application. They let you split the UI into independent, reusable pieces and think about each piece in isolation. There are two main types of components in React:

1. Functional Components:
These are JavaScript functions that accept props (input data) and return React elements to describe what should appear on the screen.
Functional components are simpler and preferred in modern React development.
They can use React Hooks (like useState, useEffect, etc.) to handle state and lifecycle methods.
Example of a functional component:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

2. Class Components:
These are ES6 classes that extend from React.Component. They have a render() method that returns React elements.
Class components also handle state and lifecycle methods directly inside the class.
Class components were more commonly used in older versions of React, but are less favored now due to the introduction of hooks in functional components.
Example of a class component:

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

Key Features of React Components:

- **Reusable:** Components can be reused in different parts of the app.
- **Isolated:** Each component manages its own state and logic, making them self-contained.
- **Composable:** Components can be composed together, meaning you can nest components inside other components to build complex UIs.
- **Stateful vs Stateless:** Functional components are considered stateless unless they use hooks to manage local state. Class components manage state directly through this.state.
- **Props:** Components receive input in the form of props (properties), which are read-only. These are used to pass data from parent to child components.

Feature	Class Component	Functional Component
Syntax	ES6 class extending `React.Component`	JavaScript function
State Management	Managed via `this.state` and `this.setState()`	Managed using hooks (`useState`)
Lifecycle Methods	Available through built-in methods	Handled via `useEffect` and other hooks
Performance	Slightly slower due to overhead of class	Generally more performant, lighter components
`this` **Context**	Requires handling of `this`	No `this` needed
Hooks	Not applicable	Can use React hooks (`useState`, `useEffect`)
Code Readability & Simplicity	More verbose	Simpler and more concise

Rendering in React refers to the process of displaying components and their content to the user interface (UI). React uses a virtual DOM to efficiently manage updates to the UI, ensuring high performance and reactivity. Here's an overview of how rendering works in React:

Types of Rendering

- Initial Rendering:
    - Happens when the application or a component is first loaded.
    - React builds the virtual DOM and calculates the changes needed to render the real DOM.
- Re-Rendering:
    - Occurs when the component's state or props change.
    - React compares the current and previous virtual DOM using a process called reconciliation and updates only the parts of the DOM that changed.

Rendering Lifecycle

- Functional Components:
    - Rendering is tied to the function call of the component.
    - Any update to state or props triggers the component to re-render.
- Class Components:
    - Controlled by the render() method.
    - The render() method is called every time setState() is used or props are updated.

Virtual DOM

- React maintains a lightweight copy of the real DOM, called the virtual DOM.
- When state or props change:
    1. React computes the updated virtual DOM.
    2. It uses diffing to identify changes between the previous and current virtual DOM.
    3. Only the necessary updates are applied to the real DOM.

This approach minimizes expensive DOM manipulation operations, enhancing performance.

Key Concepts in Rendering

- Props:
    - Data passed from parent to child components.

    - Changes in props trigger the child component to re-render.
- State:
    - Internal data managed within a component.
    - Changes to state trigger the component to re-render.
- Keys:
    - Used in lists to help React identify which items have changed, are added, or removed.
    - Improves rendering performance for lists.
- Pure Components and Memoization:
    - React.PureComponent or React.memo() can be used to prevent unnecessary re-renders by doing a shallow comparison of props and state.

Optimizing Rendering

- Avoid unnecessary renders by:
    - Using React.memo() for functional components.
    - Using shouldComponentUpdate() for class components.
    - Keeping state localized to minimize prop drilling.
- Use code-splitting and lazy loading for better performance.
- Avoid inline functions or objects in props to prevent new references during every render.

State
State is an object that represents the dynamic parts of a component. It determines how a component behaves and what it renders. It is mutable and is managed within the component.

Key Features
Local to the Component: State is owned and managed by the component itself.
Dynamic: Changes to state trigger a re-render of the component.
Initialized within the Component: State is typically initialized when the component is created.

Using State in Class Components

In class components, state is managed using the this.state object.
Example:

```
import React, { Component } from 'react';
class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 }; // Initialize state
  }

  increment = () => {
    this.setState({ count: this.state.count + 1 }); // Update state
  };

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}
export default Counter;
```

Using State in Functional Components
With the introduction of React Hooks, state can be used in functional components via the useState hook.

Example:

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

function Counter() {
  const [count, setCount] = useState(0); // Declare state

  const increment = () => {
    setCount(count + 1); // Update state
  };
```

```
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
export default Counter;
```

Updating State

State updates are asynchronous:

Use **setState** in class components.
Use setState callback for derived state:
this.setState((prevState) => ({ count: prevState.count + 1 }));

In functional components:
setCount((prevCount) => prevCount + 1);

Props
Props (short for properties) are used to pass data from a parent component to a child component. They are read-only and cannot be modified by the child component.

Key Features
Immutable: Props are immutable, meaning they cannot be changed by the receiving component.
Parent to Child Communication: Props are used to send data or functions to child components.
Stateless: Props do not maintain their own state.

Using Props
Props are passed to a child component as attributes of the component tag.
Example:

```
function Greeting({ name }) {
  return <h1>Hello, {name}!</h1>;
}

function App() {
  return <Greeting name="John" />;
}
```

Default Props
we can define default props for a component.

Example:

```
function Greeting({ name }) {
  return <h1>Hello, {name}!</h1>;
}
```

```
Greeting.defaultProps = {
  name: 'Guest', // Default value for 'name'
};

function App() {
  return <Greeting />;
}
```

State vs. Props

Feature	State	Props
Definition	Holds local, dynamic data for a component.	Passes data from parent to child.
Mutability	Mutable: Can be updated via `setState` or hooks.	Immutable: Cannot be modified.
Scope	Local to the component.	Passed from parent to child.
Purpose	For managing data that changes over time.	For passing data or functions.
Trigger Render	Yes, changes to state trigger re-renders.	Yes, when new props are passed.
Example	Counter that tracks clicks.	Passing a title to a header.

Let's combine state and props.

Parent Component:

```
function App() {
  return <Profile name="John Doe" age={30} />;
}
```

Child Component (Uses Props and State):

```
function Profile({ name, age }) {
  const [likes, setLikes] = useState(0); // Local state
  const incrementLikes = () => {
    setLikes(likes + 1); // Update state
  };

  return (
    <div>
      <h1>Name: {name}</h1>
      <h2>Age: {age}</h2>
      <p>Likes: {likes}</p>
      <button onClick={incrementLikes}>Like</button>
    </div>
  );
}
```

Note

- Use state for data that changes during the component's lifecycle.
- Use props for data or callbacks that need to be shared between components.
- Keep components as pure as possible: Minimize state in components and use props for configuration.
- Avoid directly modifying state; always use setState or useState.

In React, lifecycle methods allow developers to hook into specific points of a component's lifecycle. The lifecycle of a React component can be divided into three phases:

Mounting: When the component is being inserted into the DOM.
Updating: When the component is being re-rendered due to changes in props or state.
Unmounting: When the component is being removed from the DOM.

1. Mounting Phase

These methods are invoked when an instance of a component is being created and inserted into the DOM.

constructor(props):

Called when the component is first created. Often used to initialize state or bind event handlers.
static getDerivedStateFromProps(props, state):

Rarely used, but allows the state to be updated based on changes to props. It's invoked before rendering and also when props change during updates.

render():

Required method in class components. It reads props and state and returns the JSX (UI) that gets rendered to the DOM.

componentDidMount():

Called once the component has been rendered and inserted into the DOM. Commonly used for side effects like data fetching or subscriptions.
2. Updating Phase
These methods are called when the component is being re-rendered as a result of changes to its props or state.

static getDerivedStateFromProps(props, state):

This method is also called during the updating phase, just before rendering.
shouldComponentUpdate(nextProps, nextState):

Used to optimize re-rendering. It lets you compare the current props and state with the next ones and return true or false to control whether the component should update.

render():

Called again to update the UI based on the new props or state.

getSnapshotBeforeUpdate(prevProps, prevState):

Called right before the DOM is updated. It can capture information (like scroll position) before the DOM is actually changed.

componentDidUpdate(prevProps, prevState, snapshot):

Called after the component's updates are flushed to the DOM. It's often used for DOM updates or further side effects after changes.
3. Unmounting Phase
These methods are called when the component is being removed from the DOM.

componentWillUnmount():
Called just before the component is removed from the DOM. It's commonly used for cleanup, such as removing event listeners or cancelling network requests.

Error Handling Lifecycle Methods
These methods are called during the lifecycle in case of errors:

static getDerivedStateFromError(error):

Called when an error occurs during rendering, in a lifecycle method, or in a child component. It lets you update the state to reflect the error.

componentDidCatch(error, info):

Called after an error is thrown, allowing you to log error information or display fallback UI.

Forms and User Input

Forms in React are used to collect user input. React uses a declarative approach where form elements are controlled or uncontrolled.

Controlled Components

- In a controlled component, React state is the "single source of truth."
- An input form element's value is controlled by React via state.

Example:

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

function ControlledForm() {
  const [name, setName] = useState('');

  const handleChange = (event) => {
    setName(event.target.value); // Updates state
  };

  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`Submitted Name: ${name}`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
        <input type="text" value={name} onChange={handleChange} />
      </label>
      <button type="submit">Submit</button>
    </form>
  );
}
export default ControlledForm;
```

Uncontrolled Components

- Uncontrolled components store their values in the DOM instead of state.
- React's ref is used to access the value.

Example:

```
import React, { useRef } from 'react';
function UncontrolledForm() {
  const nameInput = useRef();

  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`Submitted Name: ${nameInput.current.value}`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
```

```
        <input type="text" ref={nameInput} />
      </label>
      <button type="submit">Submit</button>
    </form>
  );
}
export default UncontrolledForm;
```

Handling Multiple Inputs
Use state as an object and update properties based on name attributes.

```
function MultiInputForm() {
  const [formData, setFormData] = useState({ firstName: '', lastName: '' });

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

  return (
    <form>
<input name="firstName" value={formData.firstName} onChange={handleChange} />
 <input name="lastName" value={formData.lastName} onChange={handleChange} />
    </form>
  );
}
```

Event Handling
React has a unified system for handling events, where event names are written in camelCase (e.g., onClick).

Key Features
- ❖ React uses Synthetic Events (a wrapper around native browser events) for consistency across browsers.
- ❖ Event handlers in React are passed as functions, not strings.

Common Events

Event Name	Description
`onClick`	Fires when an element is clicked.
`onChange`	Fires when an input value changes.
`onSubmit`	Triggers when a form is submitted.
`onMouseEnter`	Fires when the mouse enters an element.
`onKeyDown`	Triggers when a key is pressed.

Example: Handling Clicks

```
function ButtonClick() {
  const handleClick = () => {
    alert('Button clicked!');
  };

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

Passing Arguments

To pass arguments to an event handler, use an inline function.

```
function ListItem({ id, handleRemove }) {
  return (
    <li>
      Item {id} <button onClick={() => handleRemove(id)}>Remove</button>
    </li>
  );
}
```

Preventing Default Behavior

To stop default browser actions (like a form submission):

```
function PreventDefaultForm() {
  const handleSubmit = (event) => {
    event.preventDefault(); // Stops default behavior
    alert('Form submission prevented.');
  };

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

Using Event Objects

React passes a Synthetic Event object to the handler.

```
function MouseTracker() {
  const handleMouseMove = (event) => {
    console.log(`Mouse at: (${event.clientX}, ${event.clientY})`);
  };

  return <div onMouseMove={handleMouseMove} style={{ height: '100px', background: '#eee'
}}>Move your mouse here!</div>;
}
```

Integration Example

A form with controlled components and event handling:

```
function FullForm() {
  const [formData, setFormData] = useState({ username: '', password: '' });
```

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

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

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Username:
        <input name="username" value={formData.username}
onChange={handleChange} />
      </label>
      <label>
        Password:
        <input name="password" type="password" value={formData.password}
onChange={handleChange} />
      </label>
      <button type="submit">Login</button>
    </form>
  );
}
```

Note:

- Use controlled components for better state management.
- Use ref for accessing DOM nodes in uncontrolled components.
- Use preventDefault() for overriding default behaviors (like form submissions).
- Organize event handlers outside the JSX for readability and reusability.