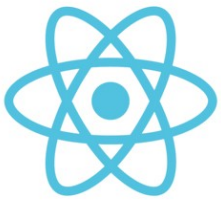
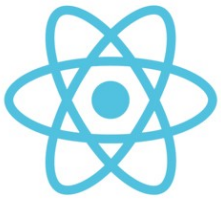


Introduction of React JS

- **Introduction :**
  - **React** is a free and open-source **front-end JavaScript library** for building **reusable** user interfaces based on components.
  - It is maintained by **Meta** (formerly Facebook) and a community of individual developers and companies.
  - It was created by **Jordan Walke**, a software engineer at the Facebook, the Person affected by XHP.
  - The React was **first deployed** for Facebook's Newsfeed application in **2011**, and then deployed for Instagram.com in **2012.**
  - It was open-sourced at JSConf US in **May 2013**.
  - React can be used to develop **single-page, mobile, or server-rendered applications** with frameworks like **Next.js**.
  - Because React is only concerned with the user interface and rendering components to the DOM, React applications often rely on libraries for routing and other client-side functionality.



Setup React Environment

1. Step 1: Install Node.js

- Node.js is a JavaScript **runtime environment** that is required to run React because React apps are built using Javascript & Node.js provides the environment to execute JS code on **the server-side.**
- so make sure you have it installed on your computer. You can download the **latest version** of Node.js from the official website: **https://nodejs.org**
- **Set Environment Path :**
    - **(C:) Drive => Program files => nodejs => (Copy path)**
    - Open **Edit system environment variables => Environment variables => Path => New => (Paste Copied path)**
    - **Restart your Device**

2. Step 2: Install VS Code

- List of popular IDEs for React Development,
    - Visual Studio Code (VS Code)
    - WebStorm
    - Atom
    - Sublime Text
- Best IDE for React development varies based on personal preference and VSCode is best.

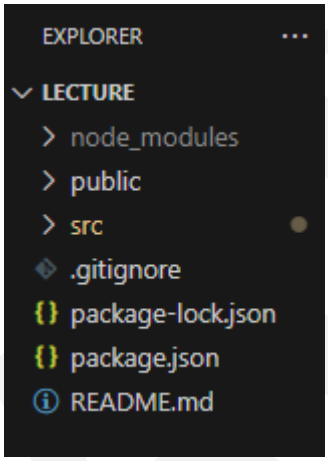
3. Step 3: Create React App

- Create React App is a comfortable environment for learning React, and is the best way to start building a new single-page application in React.

```
npx create-react-app my-app
cd my-app
npm start
```

React Folder Structure

- A typical React project follows a common folder structure that organizes the codebase.



- **src:**
  - This is the main folder where most of your project's code resides.
    - **index.js:** The entry point of your application.
    - **App.js:** The root component that gets rendered in the browser.
    - **components:** This folder contains reusable and smaller components used in the project.
    - **pages:** This folder contains larger components that represent different pages or views of your application.
    - **styles:** CSS or styling-related files, including global styles and component-specific styles.
    - **assets:** Static assets such as images, icons, or fonts used in your application.
    - **utils:** Utility functions or helper modules that are used throughout the project.
    - **services:** This folder holds any services or API-related files.

- **public:**
  - This folder contains static assets that don't require processing, such as the HTML file.

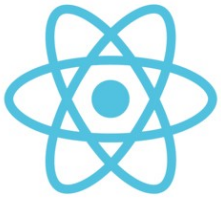
- **node_modules:**
  - It is automatically created when you install dependencies.

- **package.json:**
  - This file lists the project's metadata and dependencies, including scripts for running various tasks.

- **Package-lock.json:**
  - These files are automatically generated and lock the versions of your installed dependencies, ensuring consistent builds across different machines.

- **.gitignore:**
  - This file specifies which files or directories should be ignored by version control (e.g., Git).

- **README.md:**
  - A file containing information about the project.



React JS Code Flow

Component Rendering :

- React is a component-based library, and everything starts with components.
- You create reusable components using React's Component or Functional Component syntax.
- Components are responsible for rendering the UI and managing their own state and props.

Component Hierarchy:

- Components **can be nested** within each other, forming a component hierarchy. The parent component can pass data (props) to its child components.

State & Props:

- components can have two types of data: **state and props**.
- **State** represents the internal data of a component, and it can be changed by the component itself using setState().
- **Props** are passed down from parent components to child components and are read-only.

Rendering:

- When a component's state or props change, React triggers a re-render of that component and its child components.
- The render method of each component is called, generating a virtual representation of the component's UI.

Event Handling:

- React provides a synthetic event system that abstracts away the differences between browsers.
- You can attach event handlers to components and respond to user interactions, such as button clicks or form submissions.

Lifecycle Methods:

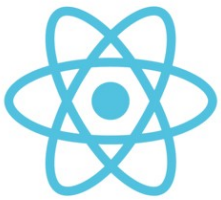
- React components have lifecycle methods that allow you to hook into different stages of a component's life, such as initialization, rendering, and unmounting.
- You can use these methods to perform actions like fetching data, subscribing to events, or cleaning up resources.

Updating State / Props:

- When a component's state or props need to be updated, you can use setState() to modify the state.
- React will then re-render the component and its children, reflecting the updated data in the UI.

Component Communication:

- Components can communicate with each other through props. Data can be passed from parent to child components, and child components can notify parent components of changes using callback function.



Components in React JS

- React components are **reusable** building blocks in the React JS library.
- They encapsulate a **piece of UI** and its associated **logic**, allowing you to compose complex user interfaces from **smaller, self-contained parts.**
- The Components can be divided into two types:
    1. **Functional** components
    2. **Class** components

1. Functional Components:

- Functional components are defined as JavaScript functions. They receive input data as properties (props) and return React elements.

```
import React from 'react'
function About(){
     return(
          <>
               <h1>Hello World...!</h1>
          </>
     )
}
export default About;
```

Using Arrow Function:

```
import React from 'react'
const About = () => {
  return (
     <h1>Hello World...!</h1>
  )
}
export default About;
```

2. Class Components:

- Class components are defined as JavaScript classes that extend the React.Component base class.
- They have more advanced features, such as local component state and lifecycle methods.
- This is no longer necessary to learn as a React developer, but it can still benefit you to learn it

```
import React from 'react';
import ReactDOM from 'react-dom';

class Test extends React.Component {
     constructor(props) {
          super(props);
          this.state = { hello: "World!" };
     }

     componentWillMount() {
          console.log("componentWillMount()");
     }

     componentDidMount() {
          console.log("componentDidMount()");
     }
```

```
	changeState() {
		this.setState({ hello: "CDMI" });
	}

	render() {
		return (
			<div>
				<h1>cdmi.in, Hello{this.state.hello}</h1>
				<h2>
					<a onClick={this.changeState.bind(this)}>Press
Here!</a>
				</h2>
			</div>);
	}

	shouldComponentUpdate(nextProps, nextState) {
		console.log("shouldComponentUpdate()");
		return true;
	}

	componentWillUpdate() {
		console.log("componentWillUpdate()");
	}

	componentDidUpdate() {
		console.log("componentDidUpdate()");
	}
}

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

JSX in React JS

- **JSX (Javascript XML)** is an extension to Javascript syntax that allows you to write **HTML-like code within your Javascript** code when working with React.
- It provides a **concise and expressive** way to describe the structure and appearance of your user interface.

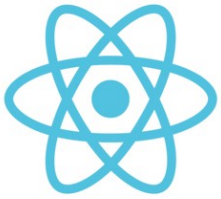
```jsx
import React from "react";

const App = () => {
  return (
    <div>
      <h1>Hello, JSX!</h1>
      <p>This is a React component.</p>
    </div>
  );
};

export default App;
```

- In this example, The code inside the return statement looks similar to HTML, but it's actually JSX.

- JSX elements can also include JavaScript **expressions** inside curly braces **{}**.
- This allows you to embed **dynamic values** or execute JavaScript code within JSX.

```jsx
const name = “Creative Design”
const greeting = <h1>Hello, {name}!</h1>;
```

- **Features of JSX :**
  - **Import** React: Begin by importing the React library.
  - One root element: JSX expressions must have a **single root element**.
  - HTML-like syntax: JSX uses **HTML**-like syntax to define elements and attributes.
  - Self-closing tags: Use **self-closing** tags for elements without children.
  - Embed JavaScript expressions within **curly braces { }.**
  - Class vs className: Use **className** instead of class for CSS classes.
  - Inline styles: Apply inline styles using the style attribute with a **JavaScript object.**
  - **Capitalize component** names to differentiate them from HTML elements.
  - Javascript expressions, not statements: Use JavaScript expressions, **not statements**, inside curly braces.
  - Commenting: Place comments **within curly braces**.
  - Fragments: Use fragments to **group elements** without adding extra nodes to the DOM.
  - Event handling: Define event handlers using **camelCase** naming conventions.
  - Conditional rendering: Use Javascript **expressions** or **logical operators** for conditional rendering.
  - JSX is not HTML: Remember that JSX is **transpiled into JavaScript**, not HTML.
  - Keys in lists: Assign **unique "key"** props to items in lists for efficient rendering.



CSS Styling React App

- In Reacts, there are several ways to apply styling to components.
- Here are some common approaches:
    1. **Inline Styles**
    2. **CSS Modules**
    3. **CSS-in-JS Libraries**
    4. **External CSS Files**

1. Inline Styles

- You can apply inline styles directly to JSX elements using the style attribute.
- The style attribute takes a JavaScript object with camel-cased CSS property-value pairs.

```
<h1 style={{backgroundColor:'yellow'}}>Creative</h1>

const styles={
     color:'blue'
};
<h1 style={styles}>Creative Design</h1>
```

2. CSS Modules

- CSS Modules allow you to write **CSS stylesheets** and **import** them as JavaScript modules.
- CSS class names are scoped locally to the component, preventing style conflicts. You can **import** the CSS file and use the defined class names in your JSX code

```
import styles from "./styles.module.css";

<div className={styles.container}>Hello</div>;
```

3. CSS-in-JS Libraries

- CSS-in-JS libraries are a popular approach for styling in React.js that allows you to write CSS directly in your JavaScript code. such as styled-components, emotion, etc..

```
import styled from "styled-components";

const StyledDiv = styled.div`
  color: red;
  font-size: 16px;
`;

<StyledDiv>Hello World</StyledDiv>;
```

4. External CSS Files

- You can also **use traditional** external CSS files and import them into your components.
- In this case, you **import the CSS file** and add the corresponding class names to your JSX elements. The styles defined in the external CSS file will be applied **globally**.

```
import "./styles.css";

<div className="container">Hello World</div>;
```

Props in React

- **What is Props?**
  - In React, props (short for **properties**) are a way to pass data from a parent
  - component to its child components, **like function arguments** in JS and **attributes** in HTML.
  - Props are an important concept in React as they enable the **flow of data** and behavior between components.
  - Props should be treated as **read-only** data, **cannot be modified**.

- **Defining Props**
  - Props are **defined as attributes** on a component when it is used in JSX.
  - For example, consider a **parent** component called Parent that renders a **child** component called Child

```
// Parent component
const Parent = () => {
  const name = "Creative Design";
  return <Child name={name} />;
};

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

- **Passing Props**
  - Props are passed from a **parent component to its child component** by specifying them as attributes on the child component's JSX tag.
  - In the **previous example**: The **name** prop is passed from the Parent component to the Child component.

```
// Parent component
    …
  const name = "Creative Design";
  <Child name={name} />;


// Child component
    …
  <h1>Hello, {props.name}!</h1>;
```

- **Receiving Props**
  - In the child component, props are received as a **single parameter.**
  - The parameter can be **named anything**, but conventionally it is named props.
  - You can access individual props using **dot notation**, such as **props.name**.

```
// Child component
    …
  <h1>Hello, {props.name}!</h1>;
```

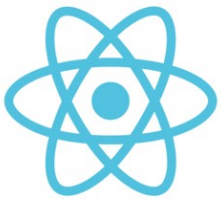
- **Dynamic Props :**
  - Props can be dynamic, meaning their values can change based on the **component's state** or **other factors**.
  - When a prop's value changes, React will **automatically re-render** the component and update the rendered output based on the new prop values.

- **Default Props :**
  - Default props provide a **fallback value** and help ensure that the component behaves as expected even if certain props are not explicitly passed.
  - In **functional** components, **default props** can be defined using the **default Props** property outside the component function.

```
const MyComponent = (props) => {
  return (
    <div>
      {props.greeting}, {props.name}!
    </div>
  );
};

MyComponent.defaultProps = {
  greeting: "Hello",
  name: "Guest"
};
```

- **Children Props :**
  - In React, the children prop is a special prop that allows you to pass **content** or **components** between the opening and closing tags of a component.
  - It provides a way to nest and compose components, **enabling more flexible** and dynamic component structures
- Passing Content as Children:
- Passing Components as Children

```jsx
// 01. Passing Content as Children:
const MyComponent = (props) => {
  return (
    <div>
      <h1>{props.title}</h1>
      {props.children}
    </div>
  );
};

const App = () => {
  return (
    <MyComponent title="Parent Component">
      <p>This is the content passed as children.</p>
    </MyComponent>
  );
};

// 2. Passing Components as Children:
const Button = (props) => {
  return <button>{props.children}</button>;
};

const App = () => {
  return (
    <Button>
      <span>Click me!</span>{" "}
    </Button>
  );
};
```



Conditional Rendering

- In React, conditional rendering is a **powerful technique** that allows you to display **different components or content** based on certain conditions.
- React provides **multiple ways** to achieve conditional rendering, depending on the complexity and requirements of your application.
    - **If-Else Statements**
    - **Ternary Operators**
    - **Logical && Operator**
    - **Conditional Render with**
    - **Components**

- **Using If-else Statements**