Why we use ReactJS?

The main objective of ReactJS is to develop User Interfaces (UI) that improves the speed of the apps. It uses virtual DOM (JavaScript object), which improves the performance of the app. The JavaScript virtual DOM is faster than the regular DOM. We can use ReactJS on the client and server-side as well as with other frameworks. It uses component and data patterns that improve readability and helps to maintain larger apps.

# **React Introduction**

ReactJS is a declarative, efficient, and flexible JavaScript library for building reusable UI components. It is an open-source, component-based front end library responsible only for the view layer of the application. It was created by **Jordan Walke,** who was a software engineer at **Facebook.** It was initially developed and maintained by Facebook and was later used in its products like **WhatsApp** & **Instagram.** Facebook developed ReactJS in **2011** in its newsfeed section, but it was released to the public in the month of **May 2013.**

Today, most of the websites are built using MVC (model view controller) architecture. In MVC architecture, React is the 'V' which stands for view, whereas the architecture is provided by the Redux or Flux.

## Why learn ReactJS?

Today, many JavaScript frameworks are available in the market(like angular, node), but still, React came into the market and gained popularity amongst them. The previous frameworks follow the traditional data flow structure, which uses the DOM (Document Object Model). DOM is an object which is created by the browser each time a web page is loaded. It dynamically adds or removes the data at the back end and when any modifications were done, then each time a new DOM is created for the same page. This repeated creation of DOM makes unnecessary memory wastage and reduces the performance of the application.

Therefore, a new technology ReactJS framework invented which remove this drawback. ReactJS allows you to divide your entire application into various components. ReactJS still used the same traditional data flow, but it is not directly operating on the browser's Document Object Model (DOM) immediately; instead, it operates on a virtual DOM. It means rather than manipulating the document in a browser after changes to our data, it resolves changes on a DOM built and run entirely in memory. After the virtual DOM has been updated, React determines what changes made to the actual browser's DOM. The React Virtual DOM exists entirely in memory and is a representation of the web browser's DOM. Due to this, when we write a React component, we did not write directly to the DOM; instead, we are writing virtual components that react will turn into the DOM.

# **React create-react-app**

Starting a new React project is very complicated, with so many build tools. It uses many dependencies, configuration files, and other requirements such as Babel, Webpack, ESLint before writing a single line of React code. Create React App CLI tool removes all that complexities and makes React app simple. For this, you need to install the package using NPM, and then run a few simple commands to get a new React project.

The **create-react-app** is an excellent tool for beginners, which allows you to create and run React project very quickly. It does not take any configuration manually. This tool is wrapping all of the required dependencies like **Webpack**, **Babel** for React project itself and then you need to focus on writing React code only. This tool sets up the development environment, provides an excellent developer experience, and optimizes the app for production.

## Installation

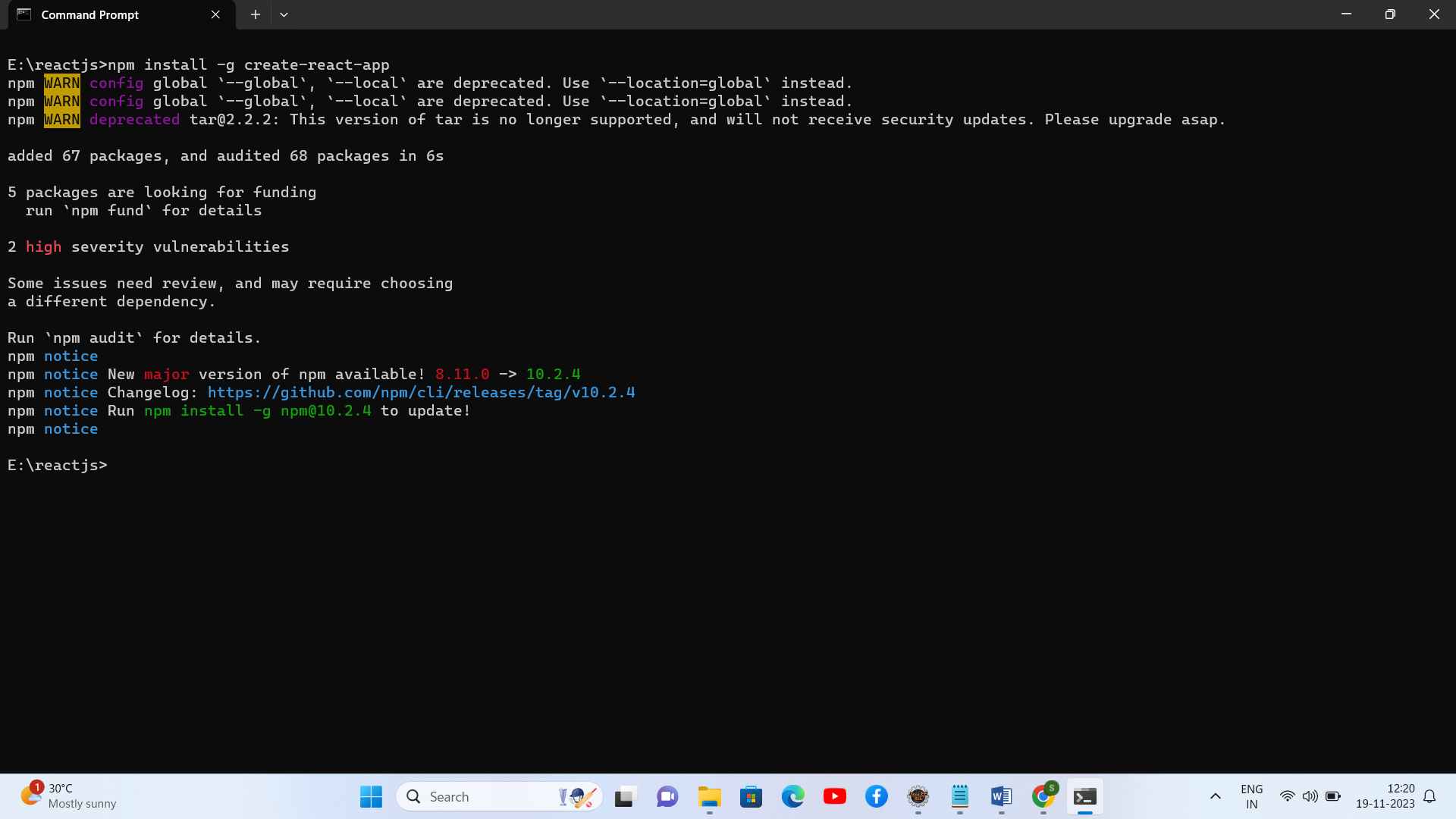
Here, we are going to learn how we can install React using **CRA** tool. For this, we need to follow the steps as given below.

### **Install React**

Step 1: Install Node Js (https://nodejs.org/en/download)

We can install React using npm package manager by using the following command. There is no need to worry about the complexity of React installation. The create-react-app npm package manager will manage everything, which needed for React project.

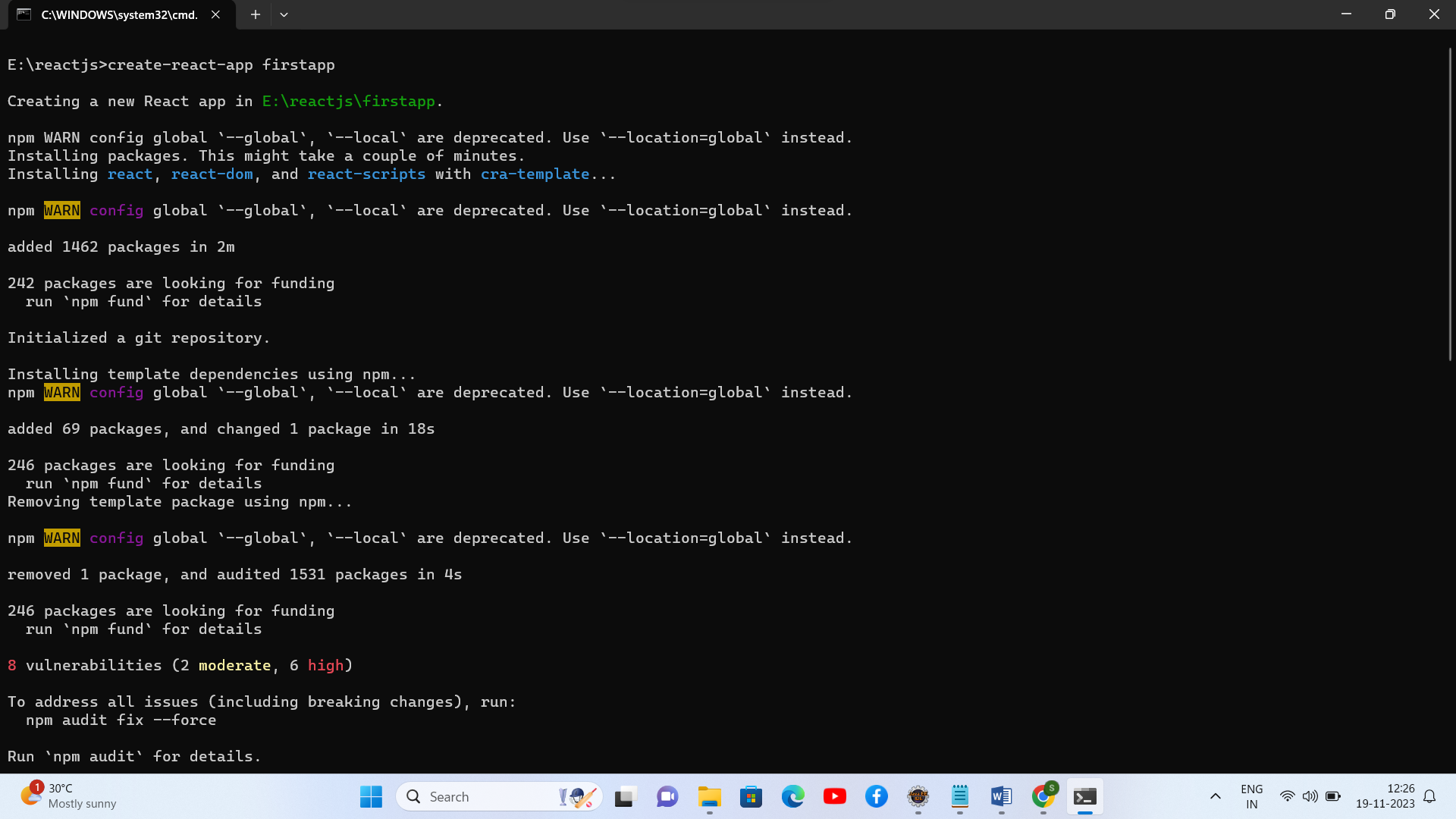
C:\> npm install -g create-react-app



### **Create a new React project**

Once the React installation is successful, we can create a new React project using create-react-app command. Here, I choose "reactproject" name for my project.

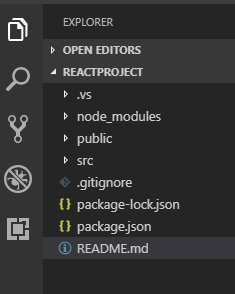
C:\> create-react-app reactproject



#### NOTE:**We can combine the above two steps in a single command using**npx**. The npx is a package runner tool which comes with npm 5.2 and above version.**

C:\ > npx create-react-app reactproject

Next, open the project on Code editor. Here, I am using Visual Studio Code. Our project's default structure looks like as below image.

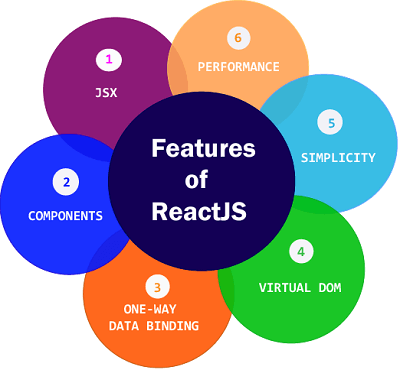


In React application, there are several files and folders in the root directory. Some of them are as follows:

1. **node\_modules:** It contains the React library and any other third party libraries needed.
2. **public:** It holds the public assets of the application. It contains the index.html where React will mount the application by default on the <div id="root"></div> element.
3. **src:** It contains the App.css, App.js, App.test.js, index.css, index.js, and serviceWorker.js files. Here, the App.js file always responsible for displaying the output screen in React.
4. **package-lock.json:** It is generated automatically for any operations where npm package modifies either the node\_modules tree or package.json. It cannot be published. It will be ignored if it finds any other place rather than the top-level package.
5. **package.json:** It holds various metadata required for the project. It gives information to npm, which allows to identify the project as well as handle the project?s dependencies.
6. **README.md:** It provides the documentation to read about React topics.

# **React Features**

----------------------------------------------



Currently, ReactJS gaining quick popularity as the best JavaScript framework among web developers. It is playing an essential role in the front-end ecosystem. The important features of ReactJS are as following.

* JSX
* Components
* One-way Data Binding
* Virtual DOM
* Simplicity
* Performance

### **JSX**

JSX stands for JavaScript XML. It is a JavaScript syntax extension. Its an XML or HTML like syntax used by ReactJS. This syntax is processed into JavaScript calls of React Framework. It extends the ES6 so that HTML like text can co-exist with JavaScript react code. It is not necessary to use JSX, but it is recommended to use in ReactJS.

### **Components**

ReactJS is all about components. ReactJS application is made up of multiple components, and each component has its own logic and controls. These components can be reusable which help you to maintain the code when working on larger scale projects.

### **One-way Data Binding**

ReactJS is designed in such a manner that follows unidirectional data flow or one-way data binding. The benefits of one-way data binding give you better control throughout the application. If the data flow is in another direction, then it requires additional features. It is because components are supposed to be immutable and the data within them cannot be changed. Flux is a pattern that helps to keep your data unidirectional. This makes the application more flexible that leads to increase efficiency.

### **Virtual DOM**

A virtual DOM object is a representation of the original DOM object. It works like a one-way data binding. Whenever any modifications happen in the web application, the entire UI is re-rendered in virtual DOM representation. Then it checks the difference between the previous DOM representation and new DOM. Once it has done, the real DOM will update only the things that have actually changed. This makes the application faster, and there is no wastage of memory.

### **Simplicity**

ReactJS uses JSX file which makes the application simple and to code as well as understand. We know that ReactJS is a component-based approach which makes the code reusable as your need. This makes it simple to use and learn.

### **Performance**

ReactJS is known to be a great performer. This feature makes it much better than other frameworks out there today. The reason behind this is that it manages a virtual DOM. The DOM is a cross-platform and programming API which deals with HTML, XML or XHTML. The DOM exists entirely in memory. Due to this, when we create a component, we did not write directly to the DOM. Instead, we are writing virtual components that will turn into the DOM leading to smoother and faster performance.

# **React JSX**

As we have already seen that, all of the React components have a **render** function. The render function specifies the HTML output of a React component. JSX(JavaScript Extension), is a React extension which allows writing JavaScript code that looks like HTML. In other words, JSX is an HTML-like syntax used by React that extends ECMAScript so that **HTML-like** syntax can co-exist with JavaScript/React code. The syntax is used by **preprocessors** (i.e., transpilers like babel) to transform HTML-like syntax into standard JavaScript objects that a JavaScript engine will parse.

JSX provides you to write HTML/XML-like structures (e.g., DOM-like tree structures) in the same file where you write JavaScript code, then preprocessor will transform these expressions into actual JavaScript code. Just like XML/HTML, JSX tags have a tag name, attributes, and children.

### **Example**

Here, we will write JSX syntax in JSX file and see the corresponding JavaScript code which transforms by preprocessor(babel).

**JSX File**

<div>Hello ReactJs</div>

**Corresponding Output**

1. React.createElement("div", **null**, "Hello ReactJs");

The above line creates a **react element** and passing **three arguments** inside where the first is the name of the element which is div, second is the **attributes** passed in the div tag, and last is the **content** you pass which is the "Hello ReactJs."

## Why use JSX?

* It is faster than regular JavaScript because it performs optimization while translating the code to JavaScript.
* Instead of separating technologies by putting markup and logic in separate files, React uses components that contain both. We will learn components in a further section.
* It is type-safe, and most of the errors can be found at compilation time.
* It makes easier to create templates.

## Nested Elements in JSX

To use more than one element, you need to wrap it with one container element. Here, we use **div** as a container element which has **three** nested elements inside it.

**App.JSX**

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

**class** App **extends** Component{

   render(){

**return**(

         <div>

            <h1>REACTJS</h1>

            <p>This website contains the best.</p>

         </div>

      );

   }

}

export **default** App;

**Output:**

## JSX Attributes

JSX use attributes with the HTML elements same as regular HTML. JSX uses **camelcase** naming convention for attributes rather than standard naming convention of HTML such as a class in HTML becomes **className** in JSX because the class is the reserved keyword in JavaScript. We can also use our own custom attributes in JSX. For custom attributes, we need to use **data- prefix**. In the below example, we have used a custom attribute **data-demoAttribute** as an attribute for the **<p>** tag.

### **Example**

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

**class** App **extends** Component{

   render(){

**return**(

         <div>

             <h1>REACTjS</h1>

   <p data-demoAttribute = "demo">This website contains the best.</p>

         </div>

      );

   }

}

export **default** App;

In JSX, we can specify attribute values in two ways:

**1 As String Literals:** We can specify the values of attributes in double quotes:

var element = <h2 className = "firstAttribute">Hello JavaReactJs</h2>;

**Example**

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

**class** App **extends** Component{

   render(){

**return**(

         <div>

            <h1 className = "hello" >HI</h1>

            <p data-demoAttribute = "demo">This website contains the best</p>

         </div>

      );

   }

}

export **default** App;

**Output:**

**2. As Expressions:** We can specify the values of attributes as expressions using curly braces {}:

var element = <h2 className = {varName}>Hello React</h2>;

**Example**

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

**class** App **extends** Component{

   render(){

**return**(

         <div>

            <h1 className = "hello" >{25+20}</h1>

         </div>

      );

   }

}

export **default** App;

**Output:**

45

## JSX Comments

JSX allows us to use comments that begin with /\* and ends with \*/ and wrapping them in curly braces {} just like in the case of JSX expressions. Below example shows how to use comments in JSX.

### **Example**

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

**class** App **extends** Component{

   render(){

**return**(

         <div>

            <h1 className = "hello" >Hello REACTJS</h1>

        {/\* This is a comment in JSX \*/}

         </div>

      );

   }

}

export **default** App;

## JSX Styling

React always recommends to use **inline** styles. To set inline styles, you need to use **camelCase** syntax. React automatically allows appending **px** after the number value on specific elements. The following example shows how to use styling in the element.

### **Example**

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

**class** App **extends** Component{

   render(){

     var myStyle = {

         fontSize: 80,

         fontFamily: 'Courier',

         color: '#003300'

      }

**return** (

         <div>

            <h1 style = {myStyle}>www.abc.com</h1>

         </div>

      );

   }

}

export **default** App;

**Output:**

#### **NOTE: JSX cannot allow to use if-else statements. Instead of it, you can use conditional (ternary) expressions. It can be seen in the following example.**

### **Example**

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

**class** App **extends** Component{

   render(){

      var i = 5;

**return** (

         <div>

            <h1>{i == 1 ? 'True!' : 'False!'}</h1>

         </div>

      );

   }

}

export **default** App;

**Output:**

False!

# **React Components**

In ReactJS, we have mainly two types of components. They are

1. Functional Components
2. Class Components

Functional Components

In React, function components are a way to write components that only contain a render method and don't have their own state. They are simply JavaScript functions that may or may not receive data as parameters. We can create a function that takes props(properties) as input and returns what should be rendered. A valid functional component can be shown in the below example.

function WelcomeMessage(props) {

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

}

The functional component is also known as a stateless component because they do not hold or manage state.

The first and recommended component type in React is functional components. A functional component is basically a JavaScript/ES6 function that returns a React element (JSX). According to React's official docs, the function below is a valid functional component:

function Welcome(props) {

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

}

Alternatively, you can also create a functional component with the arrow function definition:

const Welcome = (props) => {

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

}

This function is a valid React component because it accepts a single “props” (which stands for properties) object argument with data and returns a React element.

To be able to use a component later, you need to first export it so you can import it somewhere else:

function Welcome(props) {

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

}

export default Welcome;

After importing it, you can call the component like in this example:

import Welcome from './Welcome';

function App() {

  return (

    <div className="App">

      <Welcome name="Vijay" />

    </div>

  );

}

export default App;

So a Functional Component in React:

* is a JavaScript/ES6 function
* must return a React element (JSX)
* always starts with a capital letter (naming convention)
* takes props as a parameter if necessary

### What are Class Components?

The second type of component is the class component. Class components are ES6 classes that return JSX. Below, you see our same Welcome function, this time as a class component:

import React from "react";

class Welcome extends React.Component {

    render() {

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

    }

}

export default Welcome;

Different from functional components, class components must have an additional render( ) method for returning JSX.

### Why Use Class Components?

We used to use class components because of "state". In the older versions of React (version < 16.8), it was not possible to use state inside functional components.

Therefore, we needed functional components for rendering UI only, whereas we'd use class components for data management and some additional operations (like life-cycle methods).

This has changed with the introduction of React Hooks, and now we can also use states in functional components as well. A Class Component:

* is an ES6 class, will be a component once it ‘extends’ a React component.
* takes Props (in the constructor) if needed
* must have a render( )method for returning JSX

# **React Constructor?**

The constructor is a method used to initialize an object's state in a class. It automatically called during the creation of an object in a class.

The concept of a constructor is the same in React. The constructor in a React component is called before the component is mounted. When you implement the constructor for a React component, you need to call **super(props)** method before any other statement. If you do not call super(props) method, **this.props** will be undefined in the constructor and can lead to bugs.

### **Syntax**

Constructor(props){

**super**(props);

}

In React, constructors are mainly used for two purposes:

1. It used for initializing the local state of the component by assigning an object to this.state.
2. It used for binding event handler methods that occur in your component.

#### **Note: If you neither initialize state nor bind methods for your React component, there is no need to implement a constructor for React component.**

You cannot call **setState()** method directly in the **constructor()**. If the component needs to use local state, you need directly to use '**this.state**' to assign the initial state in the constructor. The constructor only uses this.state to assign initial state, and all other methods need to use setState() method.

### **Example**

The concept of the constructor can understand from the below example.

**App.js**

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

**class** App **extends** Component {

  constructor(props){

**super**(props);

**this**.state = {

         data: 'www.reactjs.com'

      }

**this**.handleEvent = **this**.handleEvent.bind(**this**);

  }

  handleEvent(){

    console.log(**this**.props);

  }

  render() {

**return** (

      <div className="App">

    <h2>React Constructor Example</h2>

    <input type ="text" value={**this**.state.data} />

        <button onClick={**this**.handleEvent}>Please Click</button>

      </div>

    );

  }

}

export **default** App;

**Main.js**

**import** React from 'react';

**import** ReactDOM from 'react-dom';

**import** App from './App.js';

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

**Output**

The most common question related to the constructor are:

**1 Is it necessary to have a constructor in every component?**

No, it is not necessary to have a constructor in every component. If the component is not complex, it simply returns a node.

**class** App **extends** Component {

render () {

**return** (

            <p> Name: { **this**.props.name }</p>

        );

    }

}

**2. Is it necessary to call super() inside a constructor?**

Yes, it is necessary to call super() inside a constructor. If you need to set a property or access 'this' inside the constructor in your component, you need to call super().

**class** App **extends** Component {

    constructor(props){

**this**.fName = "Jhon"; // 'this' is not allowed before super()

    }

    render () {

**return** (

            <p> Name: { **this**.props.name }</p>

        );

    }

}

When you run the above code, you get an error saying **'this' is not allowed before super()**. So if you need to access the props inside the constructor, you need to call super(props).

## Arrow Functions

The Arrow function is the new feature of the ES6 standard. If you need to use arrow functions, it is not necessary to bind any event to 'this.' Here, the scope of 'this' is global and not limited to any calling function. So If you are using Arrow Function, there is no need to bind 'this' inside the constructor.

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

**class** App **extends** Component {

  constructor(props){

**super**(props);

**this**.state = {

         data: 'www.ReactJs.com'

      }

  }

  handleEvent = () => {

    console.log(**this**.props);

  }

  render() {

**return** (

      <div className="App">

    <h2>React Constructor Example</h2>

    <input type ="text" value={**this**.state.data} />

        <button onClick={**this**.handleEvent}>Please Click</button>

      </div>

    );

  }

}

export **default** App;

We can use a constructor in the following ways:

**1) The constructor is used to initialize state.**

**class** App **extends** Component {

  constructor(props){

        // here, it is setting initial value for 'inputTextValue'

**this**.state = {

            inputTextValue: 'initial value',

        };

  }

}

**2) Using 'this' inside constructor**

**class** App **extends** Component {

    constructor(props) {

        // when you use 'this' in constructor, super() needs to be called first

**super**();

        // it means, when you want to use 'this.props' in constructor, call it as below

**super**(props);

    }

}

**3) Initializing third-party libraries**

**class** App **extends** Component {

    constructor(props) {

**this**.myBook = **new** MyBookLibrary();

        //Here, you can access props without using 'this'

**this**.Book2 = **new** MyBookLibrary(props.environment);

    }

}

**4) Binding some context(this) when you need a class method to be passed in props to children.**

**class** App **extends** Component {

    constructor(props) {

        // when you need to 'bind' context to a function

**this**.handleFunction = **this**.handleFunction.bind(**this**);

    }

}

## 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 props and state

App.js

import React, { Component } from 'react'

import './App.css'

import Greet from './components/Greet'

class App extends Component {

  render() {

    return (

      <div className="App">

        <br />

        {

<Greet name="Diana" heroName="Wonder Woman" />

}

      </div>

    )

  }

}

export default App

Greet.js

import React from 'react'

const Greet = props => {

  return (

    <div>

      <h1>

        Hello {props.name} a.k.a... {props.heroName}

      </h1>

    </div>

  )

}

export default Greet

FirstWay to Destructure

-----------------------------------

import React from 'react'

const Greet = ({name,heroName}) => {

  return (

    <div>

      <h1>

        Hello {name} a.k.a {heroName}

      </h1>

    </div>

  )

}

export default Greet

SecondWay to Destructure

---------------------------------

import React from 'react'

const Greet = props => {

  const {name,heroName}=props

  return (

    <div>

      <h1>

        Hello {name} a.k.a {heroName}

      </h1>

    </div>

  )

}

export default Greet

Destructuring Class Component

App.js

import React, { Component } from 'react'

import './App.css'

import Welcome from './components/Welcome'

class App extends Component {

  render() {

    return (

      <div className="App">

        <br />

        {

<Welcome name="Diana" heroName="Wonder Woman" />

}

      </div>

    )

  }

}

export default App

Welcome.js

import React, { Component } from 'react'

class Welcome extends Component {

  render() {

    const {name,heroName}=this.props

//const {state1,state2}= this.state

    return <h1>Welcome {name} a.k.a {heroName}</h1>

  }

}

export default Welcome

# **React Conditional Rendering**

In React, we can create multiple components which encapsulate behavior that we need. After that, we can render them depending on some conditions or the state of our application. In other words, based on one or several conditions, a component decides which elements it will return. In React, conditional rendering works the same way as the conditions work in JavaScript. We use JavaScript operators to create elements representing the current state, and then React Component update the UI to match them.

From the given scenario, we can understand how conditional rendering works. Consider an example of handling a **login/logout** button. The login and logout buttons will be separate components. If a user logged in, render the **logout component** to display the logout button. If a user not logged in, render the **login component** to display the login button. In React, this situation is called as **conditional rendering**.

There is more than one way to do conditional rendering in React. They are given below.

* if
* ternary operator
* logical && operator
* switch case operator
* Conditional Rendering with enums

## if

It is the easiest way to have a conditional rendering in React in the render method. It is restricted to the total block of the component. IF the condition is **true**, it will return the element to be rendered. It can be understood in the below example.

import React from 'react';

import ReactDOM from 'react-dom';

import './index.css';

import registerServiceWorker from './registerServiceWorker';

const UserLoggin=(props)=> {

    return <h1>Welcome back!</h1>;

  }

  function GuestLoggin(props) {

    return <h1>Please sign up.</h1>;

  }

  const SignUp=(props)=> {

    const isLoggedIn = props.isLoggedIn;

    if (isLoggedIn) {

      return <UserLoggin />;

    }

    return <GuestLoggin />;

  }

  ReactDOM.render(

    <SignUp isLoggedIn={false} />,

    document.getElementById('root')

  );

registerServiceWorker();

## Logical && operator

This operator is used for checking the condition. If the condition is **true**, it will return the element **right** after **&&**, and if it is **false**, React will **ignore** and skip it.

### **Syntax**

{

    condition &&

    // whatever written after && will be a part of output.

}

We can understand the behavior of this concept from the below example.

If you run the below code, you will see the **alert** message because the condition is matching.

(10 > 5) && alert('This alert will be shown!')

Index.js

import React from 'react';

import ReactDOM from 'react-dom';

import './index.css';

import registerServiceWorker from './registerServiceWorker';

function Example()

{

    return(<div>

            {

                (10 > 5) && alert('This alert will be shown!')

            }

           </div>

        );

}

  ReactDOM.render(

    <Example />,

    document.getElementById('root')

  );

registerServiceWorker();

## Ternary operator

The ternary operator is used in cases where two blocks alternate given a certain condition. This operator makes your if-else statement more concise. It takes **three** operands and used as a shortcut for the if statement.

### **Syntax**

condition ?  **true** : **false**

If the condition is **true**, **statement1** will be rendered. Otherwise, **false** will be rendered.

### **Example**

render() {

**const** isLoggedIn = **this**.state.isLoggedIn;

**return** (

    <div>

      Welcome {isLoggedIn ? 'Back' : 'Please login first'}.

    </div>

  );

}

## Switch case operator

Sometimes it is possible to have multiple conditional renderings. In the switch case, conditional rendering is applied based on a different state.

### **Example**

Index.js

import React from 'react';

import ReactDOM from 'react-dom';

import './index.css';

import registerServiceWorker from './registerServiceWorker';

import Dashboard from './components/Dashboard';

  ReactDOM.render(

    <Dashboard userRole='user' />,

    document.getElementById('root')

  );

registerServiceWorker();

Dashboard.js

import React from 'react';

import AdminDashboard from './AdminDashboard';

import UserDashboard from './UserDashboard';

function Dashboard(props) {

  const { userRole } = props;

  switch (userRole) {

    case 'admin':

      return <AdminDashboard />;

    case 'user':

      return <UserDashboard />;

    default:

      return <div>Error: Invalid User Role</div>;

  }

}

export default Dashboard;

AdminDashboard.js

import { Component } from "react";

class AdminDashboard extends Component

{

    render(){

            return <h1>Welcome Admin</h1>

    }

}

export default AdminDashboard;

### UserDashboard.js

import React, { Component } from "react";

class UserDashboard extends Component

{

    render(){

            return <h1>Welcome User</h1>

    }

}

export default UserDashboard;

### **Example(if,ternary,&&)**

import React, { Component } from 'react'

class UserGreeting extends Component {

  constructor(props) {

    super(props)

    this.state = {

      isLoggedIn: true

    }

  }

  // #if-else approach

  // render() {

  //   if (this.state.isLoggedIn) {

  //     return <div>Welcome Swamy</div>

  //   } else {

  //     return <div>Welcome Guest</div>

  //   }

  // }

  // #element-variables approach

  // render() {

  //   let message

  //   if (this.state.isLoggedIn) {

  //     message = <div>Welcome Swamy</div>

  //   } else {

  //     message = <div>Welcome Guest</div>

  //   }

  //   return <div>{message}</div>

  // }

  // #ternary-operator-approach

  // render() {

  //   return this.state.isLoggedIn ? (

  //     <div>Welcome Swamy</div>

  //   ) : (

  //     <div>Welcome Guest</div>

  //   )

  // }

  // #short-circuit-operator-approach

  render() {

    return this.state.isLoggedIn && <div>Welcome Swamy</div>

  }

}

export default UserGreeting

React---TO-SpringBoot

In React JS, we utilize React Context to share the state from a parent component to its child components. We also enhance it with Bootstrap to give it a more appealing appearance. These are main libraries that we use to build the React app.

* react-router-dom: Used to build a single-page app that enables users to browse through different pages in the app without reloading the entire application.
* axios: Used to make API calls to the backend for creating, reading, updating, and deleting products.
* react context: To share the app state from the parent component to its child components.
* The keyword async before a function makes the function return a promise.

### **Example**

async function myFunction() {  
  return "Hello";  
}

Is the same as:

function myFunction() {  
  return Promise.resolve("Hello");  
}

The await keyword can only be used inside an async function.

The await keyword makes the function pause the execution and wait for a resolved promise before it continues:

preventDefault

It demonstrates how to add an item to a list by using a form element with input and button elements. In this case, a preventDefault is called on the event when submitting the form to **prevent a browser reload/refresh**

<Outlet>

An <Outlet> should be used in parent route elements to render their child route elements. This allows nested UI to show up when child routes are rendered. If the parent route matched exactly, it will render a child index route or nothing if there is no index route.

# useContext

useContext is a React Hook that lets you read and subscribe to [context](https://react.dev/learn/passing-data-deeply-with-context) from your component.

const value = useContext(SomeContext)

useEffect

useEffect is a React Hook that lets you synchronize a component with an external system.

useEffect(setup, dependencies?)

useNavigate()

The useNavigate hook returns a function that lets you navigate programmatically

* Either pass a To value (same type as <Link to>) with an optional second options argument (similar to the props you can pass to [<Link>](https://reactrouter.com/en/main/components/link)), or
* Pass the delta you want to go in the history stack. For example, navigate(-1) is equivalent to hitting the back button

useRef()

The useRef Hook allows you to persist values between renders.

It can be used to store a mutable value that does not cause a re-render when updated.

It can be used to access a DOM element directly.

useParams

React JS useParams Hook helps to access the parameters of the current route to manage the dynamic routes in the URL. The react-router-dom package has useParams hooks that let you access and use the parameters of the current route as required.

# **React Events**

Just like HTML DOM events, React can perform actions based on user events.

React has the same events as HTML: click, change, mouseover etc.

## Adding Events

React events are written in camelCase syntax:

onClick instead of onclick.

React event handlers are written inside curly braces:

onClick={shoot}  instead of onclick="shoot()".

### **React:**

<button onClick={shoot}>Take the Shot!</button>

### **HTML:**

<button onclick="shoot()">Take the Shot!</button>

### **Example:**

Put the shoot function inside the Football component:

function Football() {

const shoot = () => {

alert("Great Shot!");

}

return (

<button onClick={shoot}>Take the shot!</button>

);

}

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

root.render(<Football />);

## Passing Arguments

To pass an argument to an event handler, use an arrow function.

### **Example:**

Send "Goal!" as a parameter to the shoot function, using arrow function:

function Football() {

const shoot = (a) => {

alert(a);

}

return (

<button onClick={() => shoot("Goal!")}>Take the shot!</button>

);

}

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

root.render(<Football />);

## React Event Object

Event handlers have access to the React event that triggered the function.

In our example the event is the "click" event.

### **Example:**

Arrow Function: Sending the event object manually:

function Football() {

const shoot = (a, b) => {

alert(b.type);

/\*

'b' represents the React event that triggered the function,

in this case the 'click' event

\*/

}

return (

<button onClick={(event) => shoot("Goal!", event)}>Take the shot!</button>

);

}

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

root.render(<Football />);

# **React Forms**

Just like in HTML, React uses forms to allow users to interact with the web page.

## Adding Forms in React

You add a form with React like any other element:

### **Example:**

Add a form that allows users to enter their name:

function MyForm() {

return (

<form>

<label>Enter your name:

<input type="text" />

</label>

</form>

)

}

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

root.render(<MyForm />);

This will work as normal, the form will submit and the page will refresh.

But this is generally not what we want to happen in React.

We want to prevent this default behavior and let React control the form.

## Handling Forms

Handling forms is about how you handle the data when it changes value or gets submitted.

In HTML, form data is usually handled by the DOM.

In React, form data is usually handled by the components.

When the data is handled by the components, all the data is stored in the component state.

You can control changes by adding event handlers in the onChange attribute.

We can use the useState Hook to keep track of each inputs value and provide a "single source of truth" for the entire application.

### **Example:**

Use the useState Hook to manage the input:

import { useState } from 'react';

import ReactDOM from 'react-dom/client';

function MyForm() {

const [name, setName] = useState("");

return (

<form>

<label>Enter your name:

<input

type="text"

value={name}

onChange={(e) => setName(e.target.value)}

/>

</label>

</form>

)

}

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

root.render(<MyForm />);

## Submitting Forms

You can control the submit action by adding an event handler in the onSubmit attribute for the <form>:

### **Example:**

Add a submit button and an event handler in the onSubmit attribute:

import { useState } from 'react';

import ReactDOM from 'react-dom/client';

function MyForm() {

const [name, setName] = useState("");

const handleSubmit = (event) => {

event.preventDefault();

alert(`The name you entered was: ${name}`)

}

return (

<form onSubmit={handleSubmit}>

<label>Enter your name:

<input

type="text"

value={name}

onChange={(e) => setName(e.target.value)}

/>

</label>

<input type="submit" />

</form>

)

}

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

root.render(<MyForm />);

## Multiple Input Fields

You can control the values of more than one input field by adding a name attribute to each element.

We will initialize our state with an empty object.

To access the fields in the event handler use the event.target.name and event.target.value syntax.

To update the state, use square brackets [bracket notation] around the property name.

### **Example:**

Write a form with two input fields:

import { useState } from 'react';

import ReactDOM from 'react-dom/client';

function MyForm() {

const [inputs, setInputs] = useState({});

const handleChange = (event) => {

const name = event.target.name;

const value = event.target.value;

setInputs(values => ({...values, [name]: value}))

}

const handleSubmit = (event) => {

event.preventDefault();

alert(inputs);

}

return (

<form onSubmit={handleSubmit}>

<label>Enter your name:

<input

type="text"

name="username"

value={inputs.username || ""}

onChange={handleChange}

/>

</label>

<label>Enter your age:

<input

type="number"

name="age"

value={inputs.age || ""}

onChange={handleChange}

/>

</label>

<input type="submit" />

</form>

)

}

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

root.render(<MyForm />);

**Note:** We use the same event handler function for both input fields, we could write one event handler for each, but this gives us much cleaner code and is the preferred way in React.

## Textarea

The textarea element in React is slightly different from ordinary HTML.

In HTML the value of a textarea was the text between the start tag <textarea> and the end tag </textarea>.

<textarea>

Content of the textarea.

</textarea>

In React the value of a textarea is placed in a value attribute. We'll use the useState Hook to manage the value of the textarea:

### **Example:**

A simple textarea with some content:

import { useState } from 'react';

import ReactDOM from 'react-dom/client';

function MyForm() {

const [textarea, setTextarea] = useState(

"The content of a textarea goes in the value attribute"

);

const handleChange = (event) => {

setTextarea(event.target.value)

}

return (

<form>

<textarea value={textarea} onChange={handleChange} />

</form>

)

}

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

root.render(<MyForm />);

## Select

A drop down list, or a select box, in React is also a bit different from HTML.

in HTML, the selected value in the drop down list was defined with the selected attribute:

### **HTML:**

<select>

<option value="Ford">Ford</option>

<option value="Volvo" selected>Volvo</option>

<option value="Fiat">Fiat</option>

</select>

In React, the selected value is defined with a value attribute on the select tag:

### **Example:**

A simple select box, where the selected value "Volvo" is initialized in the constructor:

function MyForm() {

const [myCar, setMyCar] = useState("Volvo");

const handleChange = (event) => {

setMyCar(event.target.value)

}

return (

<form>

<select value={myCar} onChange={handleChange}>

<option value="Ford">Ford</option>

<option value="Volvo">Volvo</option>

<option value="Fiat">Fiat</option>

</select>

</form>

)

}

By making these slight changes to <textarea> and <select>, React is able to handle all input elements in the same way.

# **React Router**

Create React App doesn't include page routing.

React Router is the most popular solution.

## Add React Router

To add React Router in your application, run this in the terminal from the root directory of the application:

npm i -D react-router-dom

**Note:** This tutorial uses React Router v6.

If you are upgrading from v5, you will need to use the @latest flag:

npm i -D react-router-dom@latest

## Folder Structure

To create an application with multiple page routes, let's first start with the file structure.

Within the src folder, we'll create a folder named pages with several files:

src\pages\:

* Layout.js
* Home.js
* Blogs.js
* Contact.js
* NoPage.js

Each file will contain a very basic React component.

## Basic Usage

Now we will use our Router in our index.js file.

### **Example**

Use React Router to route to pages based on URL:

index.js:

import ReactDOM from "react-dom/client";

import { BrowserRouter, Routes, Route } from "react-router-dom";

import Layout from "./pages/Layout";

import Home from "./pages/Home";

import Blogs from "./pages/Blogs";

import Contact from "./pages/Contact";

import NoPage from "./pages/NoPage";

export default function App() {

return (

<BrowserRouter>

<Routes>

<Route path="/" element={<Layout />}>

<Route index element={<Home />} />

<Route path="blogs" element={<Blogs />} />

<Route path="contact" element={<Contact />} />

<Route path="\*" element={<NoPage />} />

</Route>

</Routes>

</BrowserRouter>

);

}

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

root.render(<App />);

## Example Explained

We wrap our content first with <BrowserRouter>.

Then we define our <Routes>. An application can have multiple <Routes>. Our basic example only uses one.

<Route>s can be nested. The first <Route> has a path of / and renders the Layout component.

The nested <Route>s inherit and add to the parent route. So the blogs path is combined with the parent and becomes /blogs.

The Home component route does not have a path but has an index attribute. That specifies this route as the default route for the parent route, which is /.

Setting the path to \* will act as a catch-all for any undefined URLs. This is great for a 404 error page.

## Pages / Components

The Layout component has <Outlet> and <Link> elements.

The <Outlet> renders the current route selected.

<Link> is used to set the URL and keep track of browsing history.

Anytime we link to an internal path, we will use <Link> instead of <a href="">.

The "layout route" is a shared component that inserts common content on all pages, such as a navigation menu.

Layout.js:

import { Outlet, Link } from "react-router-dom";

const Layout = () => {

return (

<>

<nav>

<ul>

<li>

<Link to="/">Home</Link>

</li>

<li>

<Link to="/blogs">Blogs</Link>

</li>

<li>

<Link to="/contact">Contact</Link>

</li>

</ul>

</nav>

<Outlet />

</>

)

};

export default Layout;

Home.js:

const Home = () => {

return <h1>Home</h1>;

};

export default Home;

Blogs.js:

const Blogs = () => {

return <h1>Blog Articles</h1>;

};

export default Blogs;

Contact.js:

const Contact = () => {

return <h1>Contact Me</h1>;

};

export default Contact;

NoPage.js:

const NoPage = () => {

return <h1>404</h1>;

};

export default NoPage;

# **React useEffect Hooks**

The useEffect Hook allows you to perform side effects in your components.

Some examples of side effects are: fetching data, directly updating the DOM, and timers.

useEffect accepts two arguments. The second argument is optional.

useEffect(<function>, <dependency>)

### **Example:**

Here is an example of a useEffect Hook that is dependent on a variable. If the count variable updates, the effect will run again:

import { useState, useEffect } from "react";

import ReactDOM from "react-dom/client";

function Counter() {

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

  const [calculation, setCalculation] = useState(0);

  useEffect(() => {

    setCalculation(() => count \* 2);

  }, [count]); // <- add the count variable here

  return (

    <>

      <p>Count: {count}</p>

      <button onClick={() => setCount((c) => c + 1)}>+</button>

      <p>Calculation: {calculation}</p>

    </>

  );

}

export default Counter;

## React Context

React Context is a way to manage state globally.

It can be used together with the useState Hook to share state between deeply nested components more easily than with useState alone.

### **Use the useContext Hook**

In order to use the Context in a child component, we need to access it using the useContext Hook.

First, include the useContext in the import statement:

eimport { useState, createContext, useContext } from "react";

import ReactDOM from "react-dom/client";

const UserContext = createContext();

function Component1() {

  const [user, setUser] = useState("Jesse Hall");

  return (

    <UserContext.Provider value={user}>

      <h1>{`Hello ${user}!`}</h1>

      <Component2 />

    </UserContext.Provider>

  );

}

function Component2() {

  return (

    <>

      <h1>Component 2</h1>

      <Component3 />

    </>

  );

}

function Component3() {

  return (

    <>

      <h1>Component 3</h1>

      <Component4 />

    </>

  );

}

function Component4() {

  return (

    <>

      <h1>Component 4</h1>

      <Component5 />

    </>

  );

}

function Component5() {

  const user = useContext(UserContext);

  return (

    <>

      <h1>Component 5</h1>

      <h2>{`Hello ${user} again!`}</h2>

    </>

  );

}

export default Component1;

## What is Node.js

Node.js is a cross-platform runtime environment and library for running JavaScript applications outside the browser. It is used for creating server-side and networking web applications.

## Features of Node.js

Following is a list of some important features of Node.js that makes it the first choice of software architects.

1. **Extremely fast:**Node.js is built on Google Chrome's V8 JavaScript Engine, so its library is very fast in code execution.
2. **I/O is Asynchronous and Event Driven:**All APIs of Node.js library are asynchronous i.e. non-blocking. So a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call. It is also a reason that it is very fast.
3. **Single threaded:**Node.js follows a single threaded model with event looping.
4. **Highly Scalable:**Node.js is highly scalable because event mechanism helps the server to respond in a non-blocking way.
5. **No buffering:**Node.js cuts down the overall processing time while uploading audio and video files. Node.js applications never buffer any data. These applications simply output the data in chunks.
6. **Open source:**Node.js has an open source community which has produced many excellent modules to add additional capabilities to Node.js applications.
7. **License:**Node.js is released under the MIT license.

# **Node.js REPL**

The term REPL stands for **Read Eval Print**and**Loop**. It specifies a computer environment like a window console or a Unix/Linux shell where you can enter the commands and the system responds with an output in an interactive mode.

## REPL Environment

The Node.js or node come bundled with REPL environment. Each part of the REPL environment has a specific work.

**Read:** It reads user's input; parse the input into JavaScript data-structure and stores in memory.

**Eval:**It takes and evaluates the data structure.

**Print:**It prints the result.

**Loop:** It loops the above command until user press ctrl-c twice.

## How to start REPL

You can start REPL by simply running "node" on the command prompt

## Node.js web-based Example

A node.js web application contains the following three parts:

1. **Import required modules:** The "require" directive is used to load a Node.js module.
2. **Create server:**You have to establish a server which will listen to client's request similar to Apache HTTP Server.
3. **Read request and return response:** Server created in the second step will read HTTP request made by client which can be a browser or console and return the response.

**How to create node.js web applications**

Follow these steps:

1. **Import required module:**The first step is to use ?require? directive to load http module and store returned HTTP instance into http variable. For example:
   1. var http = require("http");
2. **Create server:**In the second step, you have to use created http instance and call http.createServer() method to create server instance and then bind it at port 8081 using listen method associated with server instance. Pass it a function with request and response parameters and write the sample implementation to return "Hello World". For example:

var http = require("http");

http.createServer(function (request, response) {

// Send the HTTP header

   // HTTP Status: 200 : OK

   // Content Type: text/plain

   response.writeHead(200, {'Content-Type': 'text/plain'});

   // Send the response body as "Hello World"

   response.end('Hello World\n');

}).listen(8081);

// Console will print the message

console.log('Server running at http://127.0.0.1:8081/');

# **Node.js Errors**

The Node.js applications generally face four types of errors:

* **Standard JavaScript errors** i.e. <EvalError>, <SyntaxError>, <RangeError>, <ReferenceError>, <TypeError>, <URIError> etc.
* **System errors**
* **User-specified errors**
* **Assertion errors**

## Node.js Errors Example 1

Let's take an example to deploy standard JavaScript error - ReferenceError.

*File: ab.js*

// Throws with a ReferenceError because b is undefined

**try** {

**const** a = 1;

**const** c = a + b;

} **catch** (err) {

  console.log(err);

}

# **Node.js File System (FS)**

In Node.js, file I/O is provided by simple wrappers around standard POSIX functions. Node File System (fs) module can be imported using following syntax:

**Syntax:**

var fs = require("fs")

## Node.js FS Reading File

Every method in fs module has synchronous and asynchronous forms.

Asynchronous methods take a last parameter as completion function callback. Asynchronous method is preferred over synchronous method because it never blocks the program execution where as the synchronous method blocks.

var fs = require("fs");

// Asynchronous read

fs.readFile('Home.js', function (err, data) {

   if (err) {

       return console.error(err);

   }

   console.log("Asynchronous read: " + data.toString());

});

// Synchronous read

var data = fs.readFileSync('Home.js');

console.log("Synchronous read: " + data.toString());

console.log("Program Ended");

# **Node.js Crypto**

The Node.js Crypto module supports cryptography. It provides cryptographic functionality that includes a set of wrappers for open SSL's hash HMAC, cipher, decipher, sign and verify functions.

## What is Hash

A hash is a fixed-length string of bits i.e. procedurally and deterministically generated from some arbitrary block of source data.

## What is HMAC

HMAC stands for Hash-based Message Authentication Code. It is a process for applying a hash algorithm to both data and a secret key that results in a single final hash.

## Encryption Example using Hash and HMAC

const crypto = require('crypto');

const secret = 'abcdefg';

const hash = crypto.createHmac('sha256', secret)

                   .update('Welcome ')

                   .digest('hex');

console.log(hash);

## Encryption example using Cipher

*File: ab.js*

**const** crypto = require('crypto');

**const** cipher = crypto.createCipher('aes192', 'a password');

var encrypted = cipher.update('Hello', 'utf8', 'hex');

encrypted += cipher.**final**('hex');

console.log(encrypted);

# **Node.js Callbacks**

Callback is an asynchronous equivalent for a function. It is called at the completion of each task. In Node.js, callbacks are generally used. All APIs of Node are written in a way to supports callbacks. For example: when a function start reading file, it returns the control to execution environment immediately so that the next instruction can be executed.

In Node.js, once file I/O is complete, it will call the callback function. So there is no blocking or wait for File I/O. This makes Node.js highly scalable, as it can process high number of request without waiting for any function to return result.

## Blocking Code Example

Follow these steps:

1. Create a text file named **input.txt**having the following content:

Java is a very simple language.

1. Create a JavaScript file named **main.js** having the following code:

var fs = require("fs");

var data = fs.readFileSync('Home.js');

console.log(data.toString());

console.log("Program Ended");

## Non Blocking Code Example

Follow these steps:

Create a text file named **input.txt**having the following content:

Java is a very simple language.

Create a JavaScript file named **main.js** having the following code:

var fs = require("fs");

fs.readFile('Home.js', function (err, data) {

    if (err) return console.error(err);

    console.log(data.toString());

});

console.log("Program Ended");