

# React Introduction

**ReactJS**, also known as **React**, is a popular JavaScript library for building user interfaces. It is also referred to as a front-end JavaScript library.

## What is React ?

**React** is a **JavaScript library** for building **user interfaces** (UIs) on the web. React is a declarative, component based library that allows developers to build reusable UI components and It follows the Virtual DOM (Document Object Model) approach, which optimises rendering performance by minimising DOM updates. React is **fast** and works well with other tools and libraries.

## Prerequisite of React

For learning React first you have a clear understanding of HTML, CSS and JavaScript.

- HTML and CSS
- JavaScript and ES6
- JSX (Javascript XML) & Babel
- Node+Npm
- Git and CLI (Command Line Interface).

## History of React

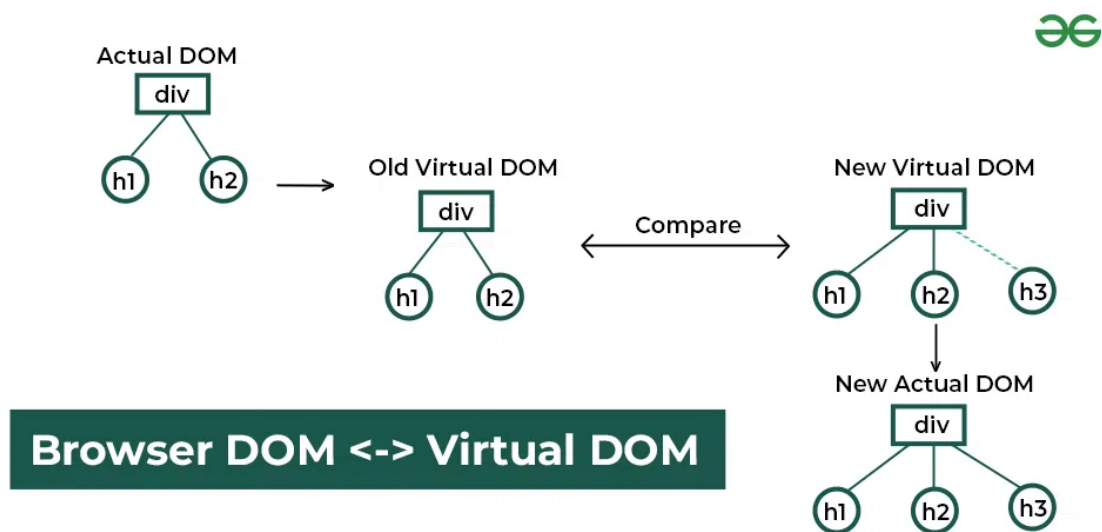
- React was invented by Facebook developers who found the traditional DOM slow. By implementing a virtual DOM, React addressed this issue and

gained popularity rapidly.

- The current stable version of ReactJS is 18.2.0, released on June 14, 2022. The library continues to evolve, introducing new features with each update.

## How does React work?

React operates by creating an in-memory virtual DOM rather than directly manipulating the browser's DOM. It performs necessary manipulations within this virtual representation before applying changes to the actual browser DOM. React is efficient, altering only what requires modification.



## Features of React:

React is one of the most demanding JavaScript libraries because it is equipped with a ton of features which makes it faster and production-ready. Below are the few features of React.

## 1. Component-Based Architecture

React provides the feature to break down the UI into smaller, self-contained components. Each component can have its own **state and props**.

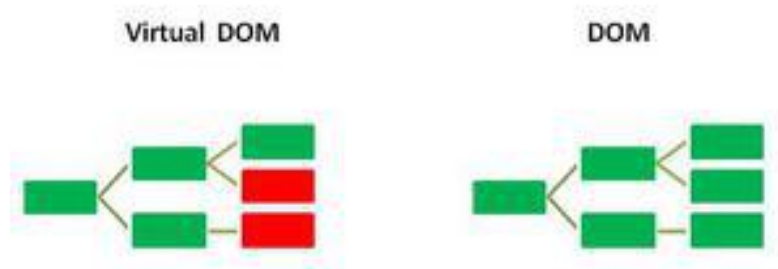
## 2. JSX (JavaScript Syntax Extension)

JSX is a syntax extension for JavaScript that allows developers to write HTML-like code within their JavaScript files. It makes React components more readable and expressive.

```
const name="hello";  
const ele = <h1>Welcome to {name}</h1>;
```

## 3. Virtual DOM

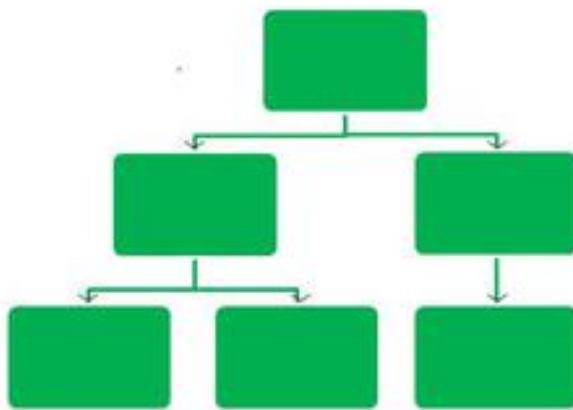
React maintains a lightweight representation of the actual DOM in memory. When changes occur, React efficiently updates only the necessary parts of the DOM.



## 4. One-way Data Binding

One-way data binding, the name itself says that it is a one-direction flow. The data in react flows only in one

direction i.e. the data is transferred from top to bottom i.e. from parent components to child components. The properties(props) in the child component cannot return the data to its parent component but it can have communication with the parent components to modify the states according to the provided inputs.



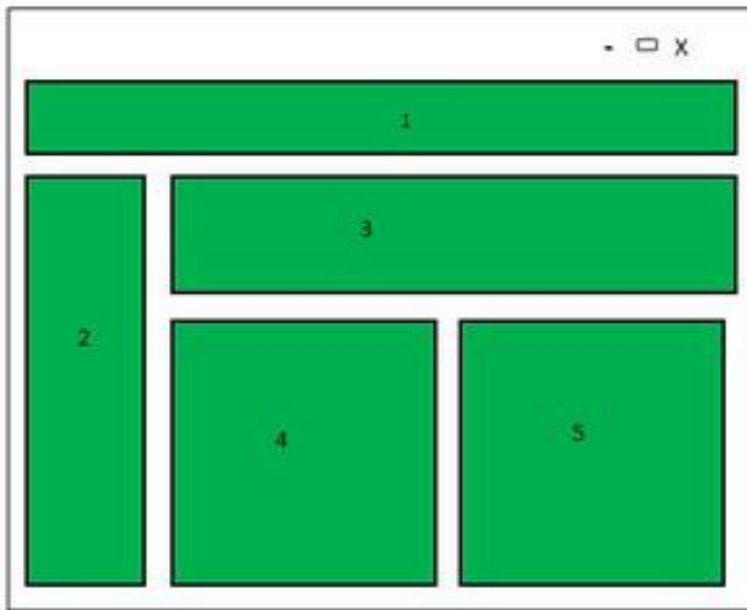
## 5. Performance

As we discussed earlier, react uses virtual DOM and updates only the modified parts. So , this makes the DOM to run faster. DOM executes in memory so we can create separate components which makes the DOM run faster.

## 6. Components

React divides the web page into multiple components as it is component-based. Each component is a part of the UI design which has its own logic and design as shown in the below image. So the component logic which is

written in JavaScript makes it easy and run faster and can be reusable.



## 7. Single-Page Applications (SPAs)

React is recommended in creating SPAs, allowing smooth content updates without page reloads. Its focus on reusable components makes it ideal for real-time applications.

## ReactJS Lifecycle:

Every React Component has a lifecycle of its own, lifecycle of a component can be defined as the series of methods that are invoked in different stages of the component's existence. React automatically calls these methods at different points in a component's life cycle. Understanding these phases helps manage state, perform side effects, and optimize components

effectively.

## 1. Initialization

This is the stage where the component is constructed with the given Props and default state. This is done in the constructor of a Component Class.

## 2. Mounting Phase

- **Constructor:** The constructor method initializes the component. It's where you set up initial state and bind event handlers.
- **render():** This method returns the JSX representation of the component. It's called during initial rendering and subsequent updates.
- **componentDidMount():** After the component is inserted into the DOM, this method is invoked. Use it for side effects like data fetching or setting timers.

## 3. Updating Phase

- **componentDidUpdate(prevProps, prevState):** Called after the component updates due to new props or state changes. Handle side effects here.
- **shouldComponentUpdate(nextProps, nextState):** Determines if the component should re-render. Optimize performance by customizing this method.
- **render():** Again, the render() method reflects changes in state or props during updates.

## 4. Unmounting Phase

- **componentWillUnmount():** Invoked just before the component is removed from the DOM. Clean up resources (e.g., event listeners, timers).

## FAQs on React

### Is React a framework or library?

It is very confusing to a lot of people that if React is a framework or a library. React is considered as a library rather than a framework. While in the framework there is a controlled way of structure to write the code whether in library you are free to write without any structural restriction.

### What is JSX?

JSX, which stands for JavaScript XML, is a syntax extension for JavaScript. ReactJS uses an XML or HTML-like syntax, which is then transformed into React Framework JavaScript calls. Essentially, JSX expands ES6 to allow HTML-like text to coexist with JavaScript React code. Although it is not mandatory to use JSX in ReactJS, it is highly recommended.

### Syntax:

```
const example = "JSX"
```

```
const ele = <div>This component uses {example} </div>
```

## Is React Beginner Friendly?

Yes, React is very simple and beginner friendly. It just uses JavaScript and JSX to create the Single page Application. If you have the basic knowledge of HTML, CSS and JavaScript you are good to go to dive deep into the React world.

## React Environment Setup

To run any React application, we need to first setup a ReactJS Development Environment.

### Table of Content

- [Using create-react-app \(CRA command\)](#)
- [Using webpack and babel](#)
- [Using Vite build tool](#)

### Pre-requisite:

We must have NodeJS installed on our PC. So, the very first step will be to install NodeJS. Once we have set up NodeJS on our PC, the next thing we need to do is set up React Boilerplate.

### Method 1: Using create-react-app (CRA command)

**Step 1:** Navigate to the folder where you want to create the project and open it in terminal



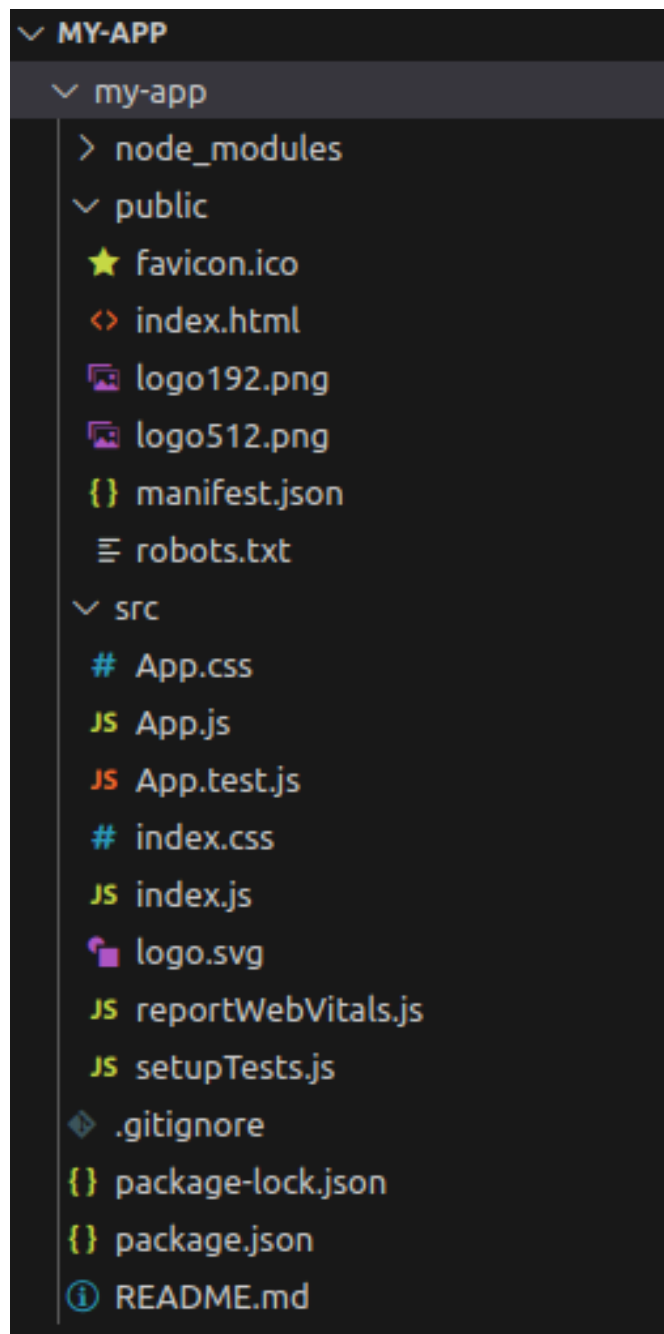
**Step 2:** In the terminal of the application directory type the following command

```
npx create-react-app <<Application_Name>>
```

**Step 3:** Navigate to the newly created folder using the command

```
cd <<Application_Name>>
```

**Step 4:** A default application will be created with the following project structure and dependencies



Application structure

It will install some packages by default which can be seen in the dependencies in package.json file as follows:

```
"dependencies": {  
  "@testing-library/jest-dom": "^5.17.0",  
  "@testing-library/react": "^13.4.0",  
  "@testing-library/user-event": "^13.5.0",  
  "react": "^18.2.0",  
}
```

```
"react-dom": "^18.2.0",  
"react-scripts": "5.0.1",  
"web-vitals": "^2.1.4"  
}
```

**Step 5:** To run this application type the following command in terminal

```
npm start
```

## React JSX

React JSX is a syntax extension of JavaScript for writing React Code in a simple way. Using JSX it is easier to create reusable UI components with fewer lines of code in a template-type language with the power of JavaScript.

### Table of Content

- [What is JSX ?](#)
- [Why JSX ?](#)
- [Expressions in JSX](#)
- [Attributes in JSX](#)
- [Specifying attribute values](#)
- [Wrapping elements or Children in JSX](#)
- [Comments in JSX](#)
- [Converting HTML to JSX](#)
- [Rules to Write JSX](#)

### What is JSX ?

**JSX** stands for **JavaScript XML**. JSX is basically a syntax extension of JavaScript.

React JSX helps us to write HTML in JavaScript and forms the basis of React Development. Using JSX is not compulsory but it is highly recommended for programming in React as it makes the development process easier as the code becomes easy to write and read.

JSX creates an element in React that gets rendered in the UI. It is transformed into JavaScript functions by the compiler at runtime. Error handling and warnings become easier to handle when using JSX

### **React JSX sample code:**

```
const ele = <h1>This is sample JSX</h1>;
```

This is called JSX (JavaScript XML), it somewhat looks like HTML and also uses a JavaScript-like variable but is neither HTML nor JavaScript. With the help of JSX, we have directly written the HTML syntax in JavaScript.

## **Why JSX ?**

- It is faster than normal JavaScript as it performs optimisations while translating to regular JavaScript.
- It makes it easier for us to create templates.
- Instead of separating the markup and logic in separate files, React uses components for this purpose.
- As JSX is an expression, we can use it inside of if statements and for loops, assign it to variables, accept it as arguments, or return it from functions.

# Expressions in JSX

In React we are allowed to use normal JavaScript expressions with JSX. To embed any JavaScript expression in a piece of code written in JSX we will have to wrap that expression in curly braces {}. The below example specifies a basic use of JavaScript Expression in React.

## Syntax:

```
const example = "JSX"
const ele = <div>This component uses {example}
</div>
```

**Example 1:** This example wraps the JSX code in curly braces

```
// Filename - App.js

import React from "react";

const name = "Learner";

const element = (
  <h1>
    Hello,
    {name}.Welcome to hello.
  </h1>
);

ReactDOM.render(element,
document.getElementById("root"));
```

## Output:

In the above program, we have embedded the javascript

expression `const name = "Learner";` in our JSX code. We can use conditional statements instead of if-else statements in JSX.

**Example 2:** In this example where conditional expression is embedded in JSX:

```
// Filename - App.js

import React from "react";

let i = 1;

const element = (
  <h1>{i == 1 ? "Hello World!" : "False!"} </h1>
);

ReactDOM.render(element,
  document.getElementById("root"));
```

**Output:**

**Hello World!**

In the above example, the variable 'i' is checked if for the value 1. As it equals 1 so the string 'Hello World!' is returned to the JSX code. If we modify the value of the variable 'i' then the string 'False!' will be returned.

## Attributes in JSX

JSX allows us to use attributes with the HTML elements just like we do with normal HTML. But instead of the normal naming convention of HTML, JSX uses the camelcase convention for attributes.&

- **The change of class attribute to className:**The class in HTML becomes className in JSX. The main reason behind this is that some attribute names in HTML like 'class' are reserved keywords in JavaScript. So, in order to avoid this problem, JSX uses the camel case naming convention for attributes.
- **Creation of custom attributes:**We can also use custom attributes in JSX. For custom attributes, the names of such attributes should be prefixed by **data-\*** attribute.

**Example:** This example has a custom attribute with the <h2> tag and we are using className attribute instead of class.

```
// Filename - App.js

import React from "react";
import ReactDOM from "react-dom";

const element = (
  <div>
    <h1 className="hello">Hello Geek</h1>
    <h2 data-sampleAttribute="sample">
      Custom attribute
    </h2>
  </div>
);

ReactDOM.render(element,
document.getElementById("root"));
```

## Specifying attribute values :

JSX allows us to specify attribute values in two ways:

- **As for string literals:** We can specify the values of attributes as hard-coded strings using quotes:  

```
const ele = <h1 className =
"firstAttribute">Hello!</h1>;
```
- **As expressions:** We can specify attributes as expressions using curly braces {}: 

```
const ele =
<h1 className = {varName}>Hello!</h1>;
```

## Wrapping elements or Children in JSX

Consider a situation where you want to render multiple tags at a time. To do this we need to wrap all of these tags under a parent tag and then render this parent element to the HTML. All the subtags are called child tags or children of this parent element.

**Example:** In this example we have wrapped h1, h2, and h3 tags under a single div element and rendered them



to HTML:

```
// Filename - App.js

import React from "react";
import ReactDOM from "react-dom";

const element = (
  <div>
    <h1>This is Heading 1 </h1>
    <h2>This is Heading 2</h2>
    <h3>This is Heading 3 </h3>
  </div>
);

ReactDOM.render(element,
document.getElementById("root"));
```

**Output:**

# This is Heading 1

## This is Heading 2

### This is Heading 3

REactJS JSX

**Note:** JSX will throw an error if the HTML is not correct or if there are multiple child elements without a parent element.

## Comments in JSX :

JSX allows us to use comments as it allows us to use JavaScript expressions. Comments in JSX begin with `/*` and ends with `*/`. We can add comments in JSX by wrapping them in curly braces `{}` just like we did in the case of expressions. The below example shows how to add comments in JSX:

```
// Filename - App.js

import React from "react";
import ReactDOM from "react-dom";

const element = (
  <div>
    <h1>Hello World !{/*This is a comment*/}</h1>
  </div>
);

ReactDOM.render(element,
  document.getElementById("root"));
```

**Output:**

**Hello World !**

## Converting HTML to JSX

Let us take the following HTML Code and Convert it into JSX

```
<!DOCTYPE html>
<html>

<head>
  <title>Basic Web Page</title>
</head>
<body>
  <h1>Welcome to hello</h1>
  <p>A computer science portal for all</p>
</body>

</html>
```

**The Converted JSX Code will look like:**

```
<>
  <title>Basic Web Page</title>
  <h1>Welcome to hello</h1>
  <p>A computer science portal for all</p>
</>
```

## Rules to Write JSX

- **Always return a single Root Element:** When there are multiple elements in a component and you want to return all of them wrap them inside a single component
- **Close all the tags:** When we write tags in HTML some of them are self closing like the <img> tag but JSX requires us to close all of them so image tag will be represented as <img />
- **Use camelCase convention wherever possible:** When writing JSX if we want to give class to a tag

we have to use the className attribute which follows camelCase convention.

## ReactJS Babel Introduction

In this article, we will discuss **Babel** and why it is one of the most widely used transpilers in the world.

### Table of Content

- What is babel?
- Using Babel with React
- Why do we need Babel?
- Features of Babel

## What is babel?

Babel is a very famous **transpiler** that basically allows us to use future JavaScript in today's browsers. In simple words, it can convert the latest version of JavaScript code into the one that the browser understands. Transpiler is a tool that is used to convert source code into another source code that is of the same level. The latest standard version that JavaScript follows is ES2020 which is not fully supported by all browsers hence we make use of a tool such as 'babel' so that we can convert it into the code that today's browser understands.

# Using Babel with React

We use Babel with React to transpile the JSX code into simple React functions that can be understood by browsers. Using this way we can assure that our JSX code can work in almost any browser. This combination is widely used in modern-day web development.

**In order to manually setup babel in React with webpack follow the below steps.**

**Step 1:** Create the folder where you want the application and navigate to it using the command:

```
mkdir my-app  
cd my-app
```

**Step 2:** After navigating to the directory type the following command

```
npm init -y
```

Make sure this is displayed in the terminal

Wrote to /home/reactapp/my-app/package.json:

```
{  
  "name": "my-app",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \"Error: no test specified\"  
&& exit 1"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC"
```

```
}
```

**Step 3:** Install the necessary react packages using the following command

```
npm i react react-dom
```

**Step 4:** Install webpack and babel using the command

```
npm i webpack webpack-cli @babel/core @babel/
preset-env @babel/preset-react babel-loader html-
webpack-plugin webpack-dev-server --save-dev
```

After following the above steps the dependencies in package.json will look like:

```
"dependencies": {
  "babel-core": "^6.26.3",
  "babel-loader": "^9.1.3",
  "babel-preset-env": "^1.7.0",
  "babel-preset-react": "^6.24.1",
  "html-webpack-plugin": "^5.5.3",
  "react": "^18.2.0",
  "react-dom": "^18.2.0",
  "webpack": "^5.88.2",
  "webpack-cli": "^5.1.4",
  "webpack-dev-server": "^4.15.1"
}
```

**Step 5:** Create the files named **index.html**, **App.js**, **index.js**, **webpack.config.js**, **.babelrc**

**Step 6:** Write the following code in webpack.config.js file

```
const path = require('path');
const HtmlWebpackPlugin = require('html-
webpack-plugin');
module.exports = {
```

```

    entry: './index.js', // Entry point of your
application
    output: {
      path: path.resolve(__dirname, 'dist'),
      filename: 'bundle.js', // Output bundle
file name
    },
    module: {
      rules: [
        {
          test: /\.js$/,
          exclude: /node_modules/,
          use: {
            loader: 'babel-loader', // Use Babel
for .js and .jsx files
          },
        },
      ],
    },
    plugins: [
      new HtmlWebpackPlugin({
        template: './index.html', // Use this
HTML file as a template
      }),
    ],
  };

```

**Step 7:** Inside the scripts section of package.json file add the following code

```

"scripts": {
  "start": "webpack-dev-server --mode
development --open",
  "build": "webpack --mode production"
}

```

**Step 8:** Add the following code in **index.html**, **index.js**, and **App.js**

```
// index.js

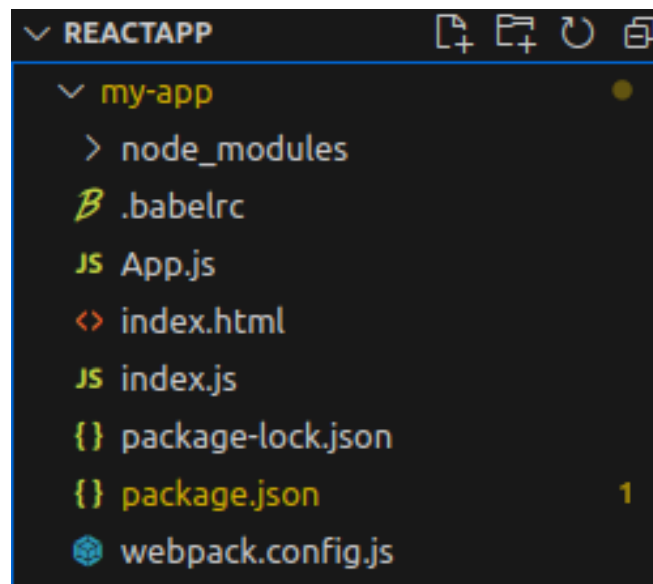
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App'; // Import your main React
component

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

**Step 9:** Inside the **.babelrc** file add the following code

```
{
  "presets": ["@babel/preset-env", "@babel/
preset-react"]
}
```

After following all the above steps the project structure should look like



Directory Structure

**The package.json should look like:**

```
{
```



```
"name": "my-app",
"version": "1.0.0",
"description": "",
"main": "index.js",
"scripts": {
  "start": "webpack-dev-server --mode
development --open",
  "build": "webpack --mode production"
},
"keywords": [],
"author": "",
"license": "ISC",
"dependencies": {
  "babel-core": "^6.26.3",
  "babel-preset-env": "^1.7.0",
  "babel-preset-react": "^6.24.1"
},
"devDependencies": {
  "@babel/core": "^7.22.9",
  "@babel/preset-env": "^7.22.9",
  "@babel/preset-react": "^7.22.5",
  "babel-loader": "^9.1.3",
  "html-webpack-plugin": "^5.5.3",
  "react": "^18.2.0",
  "react-dom": "^18.2.0",
  "webpack": "^5.88.2",
  "webpack-cli": "^5.1.4",
  "webpack-dev-server": "^4.15.1"
}
}
```

**Step 9:** To run the application type the following command in a web browser

npm start

**Output:** The output in the browser will look like

## Why do we need Babel?

The main reason we need Babel is that it gives us the privilege to make use of the latest things JavaScript has to offer without worrying about whether it will work in the browser or not.

## Features of Babel:

- **Babel-Plugins:** The Plugins are configuration details for Babel to transpile the code that supports a number of plugins, which could be used individually, provided the environment is known.
- **Babel-Presets:** Babel presets have a set of plugins that instruct Babel to transpile in a specific mode. provided the environment is known.
- **Babel-Polyfills:** During instances when methods and objects, cannot be transpiled, We can make use of babel-polyfill to facilitate the use of features in any browser.
- **Babel-CLI:** The Command-line interface of Babel has a lot of commands where the code can be easily compiled on the command line. It also has features like plugins and presets to be used along with the command making it easy to transpile the code at once.

## ReactJS Virtual DOM

React JS Virtual DOM is an in-memory representation of the DOM. DOM refers to the Document Object Model that represents the content of XML or HTML documents as a tree structure so that the programs can be read,

accessed and changed in the document structure, style, and content.

## Table of Content

- [Prerequisites](#)
- [What is DOM ?](#)
- [Disadvantages of real DOM](#)
- [Virtual DOM](#)
- [How does virtual DOM actually make things faster?](#)
- [How virtual DOM Helps React?](#)
- [Virtual DOM Key Concepts](#)
- [Differences between Virtual DOM and Real DOM](#)

## What is DOM ?

DOM stands for 'Document Object Model'. In simple terms, it is a structured representation of the HTML elements that are present in a webpage or web app. DOM represents the entire UI of your application. The DOM is represented as a tree data structure. It contains a node for each UI element present in the web document. It is very useful as it allows web developers to modify content through JavaScript, also it being in structured format helps a lot as we can choose specific targets and all the code becomes much easier to work with.

## Disadvantages of real DOM :

Every time the DOM gets updated, the updated element and its children have to be rendered again to update the UI of our page. For this, each time there is a component update, the DOM needs to be updated and the UI

components have to be re-rendered.

### **Example:**

```
// Simple getElementById() method
document.getElementById('some-id').innerHTML
= 'updated value';
```

When writing the above code in the console or in the JavaScript file, these things happen:

- The browser parses the HTML to find the node with this id.
- It removes the child element of this specific element.
- Updates the element(DOM) with the 'updated value'.
- Recalculates the CSS for the parent and child nodes.
- Update the layout.
- Finally, traverse the tree and paint it on the screen(browser) display.

Recalculating the CSS and changing the layouts involves complex algorithms, and they do affect the performance. So React has a different approach to dealing with this, as it makes use of something known as Virtual DOM.

## **Virtual DOM**

React uses Virtual DOM exists which is like a lightweight copy of the actual DOM(a virtual representation of the DOM). So for every object that exists in the original

DOM, there is an object for that in React Virtual DOM. It is exactly the same, but it does not have the power to directly change the layout of the document.

**Manipulating DOM is slow, but manipulating Virtual DOM is fast** as nothing gets drawn on the screen. So each time there is a change in the state of our application, the virtual DOM gets updated first instead of the real DOM.

## **How does virtual DOM actually make things faster?**

When anything new is added to the application, a virtual DOM is created and it is represented as a tree. Each element in the application is a node in this tree. So, whenever there is a change in the state of any element, a new Virtual DOM tree is created. This new Virtual DOM tree is then compared with the previous Virtual DOM tree and make a note of the changes. After this, it finds the best possible ways to make these changes to the real DOM. Now only the updated elements will get rendered on the page again

## **How virtual DOM Helps React?**

In React, everything is treated as a component be it a functional component or class component. A component can contain a state. Whenever the state of any component is changed react updates its Virtual DOM tree. Though it may sound like it is ineffective the cost is not much significant as updating the virtual DOM doesn't

take much time.

React maintains two Virtual DOM at each time, one contains the updated Virtual DOM and one which is just the pre-update version of this updated Virtual DOM.

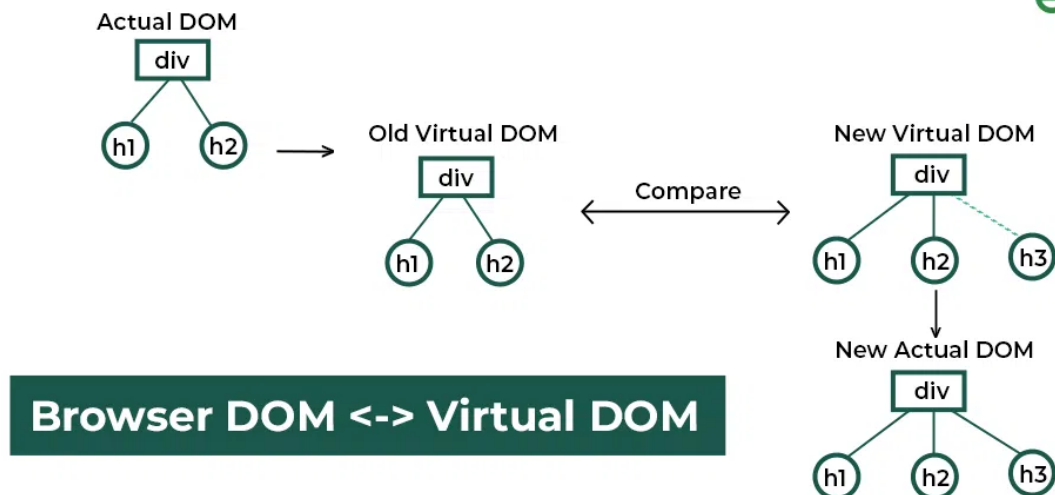
Now it compares the pre-update version with the updated Virtual DOM and figures out what exactly has changed in the DOM like which components have been changed. This process of comparing the current Virtual DOM tree with the previous one is known as 'diffing'. Once React finds out what exactly has changed then it updates those objects only, on real DOM.

React uses something called batch updates to update the real DOM. It just means that the changes to the real DOM are sent in batches instead of sending any update for a single change in the state of a component.

We have seen that the re-rendering of the UI is the most expensive part and React manages to do this most efficiently by ensuring that the Real DOM receives batch updates to re-render the UI. This entire process of transforming changes to the real DOM is called **Reconciliation**.

This significantly improves the performance and is the main reason why React and its Virtual DOM are much loved by developers all around.

The diagrammatic image below briefly describes how the virtual DOM works in the real browser environment.



Real DOM to Virtual DOM

## Virtual DOM Key Concepts :

- Virtual DOM is the virtual representation of Real DOM
- React update the state changes in Virtual DOM first and then it syncs with Real DOM
- Virtual DOM is just like a blueprint of a machine, can do changes in the blueprint but those changes will not directly apply to the machine.
- Virtual DOM is a programming concept where a virtual representation of a UI is kept in memory synced with “Real DOM ” by a library such as ReactDOM and this process is called reconciliation
- Virtual DOM makes the performance faster, not because the processing itself is done in less time. The reason is the amount of changed information – rather than wasting time on updating the entire page, you can dissect it into small elements and interactions

## Differences between Virtual DOM and Real DOM

Virtual DOM	Real DOM
It is a lightweight copy of the original DOM	It is a tree representation of HTML elements
It is maintained by JavaScript libraries	It is maintained by the browser after parsing HTML elements
After manipulation it only re-renders changed elements	After manipulation, it re-render the entire DOM
Updates are lightweight	Updates are heavyweight
Performance is fast and UX is optimised	Performance is slow and the UX quality is low
Highly efficient as it performs batch updates	Less efficient due to re-rendering of DOM after each update

## React JS ReactDOM

React JS ReactDOM or react-dom is the most widely used package of React. React provides the developers with a package **react-dom** to access and modify the DOM. Let's see in brief what is the need to have the package.

### Table of Content

- What is ReactDOM ?
- How to use ReactDOM ?
- Why ReactDOM is used ?
- Important functions provided by ReactDOM
- Key features of ReactDOM



## What is ReactDOM ?

ReactDOM is a package in React that provides DOM-specific methods that can be used at the top level of a web app to enable an efficient way of managing DOM elements of the web page. ReactDOM provides the developers with an API containing the various methods to manipulate DOM.

## How to use ReactDOM ?

To use the ReactDOM in any React web app we must first install the react-dom package in our project. To install the react-dom package use the following command.

```
// Installing  
npm i react-dom
```

After installing the package use the following command to import the package in your application file

```
// Importing  
import ReactDOM from 'react-dom'
```

After installing **react-dom** it will appear in the **dependencies** in **package.json** file like:

```
"dependencies": {  
  "react": "^18.2.0",  
  "react-dom": "^18.2.0",  
  "react-scripts": "5.0.1",  
}
```

# Why ReactDOM is used ?

Earlier, React Developers directly manipulated the DOM elements which resulted in frequent DOM manipulation, and each time an update was made the browser had to recalculate and repaint the whole view according to the particular CSS of the page, which made the total process to consume a lot of time.

To solve this issue, React brought into the scene the virtual DOM. The **Virtual DOM** can be referred to as a copy of the actual DOM representation that is used to hold the updates made by the user and finally reflect it over to the original Browser DOM at once consuming much lesser time.

## Important functions provided by ReactDOM

- **render()**: This is one of the most important methods of ReactDOM. This function is used to render a single React Component or several Components wrapped together in a Component or a div element.
- **findDOMNode()**: This function is generally used to get the DOM node where a particular React component was rendered. This method is very less used like the following can be done by adding a ref attribute to each component itself.
- **unmountComponentAtNode()**: This function is used to unmount or remove the React Component

that was rendered to a particular container.

- **hydrate()**: This method is equivalent to the render() method but is implemented while using server-side rendering.
- **createPortal()**: It allow us to render a component into a DOM node that resides outside the current DOM hierarchy of the parent component.
- 

## Key features of ReactDOM :

- ReactDOM.render() replaces the child of the given container if any. It uses a highly efficient diff algorithm and can modify any subtree of the DOM.
- React findDOMNode() function can only be implemented upon mounted components thus Functional components can not be used in findDOMNode() method.
- ReactDOM uses observables thus provides an efficient way of DOM handling.
- ReactDOM can be used on both the client-side and server-side.

## React Lists

**Lists** are very useful when it comes to developing the UI of any website. **React Lists** are mainly used for **displaying menus** on a website, for example, the **navbar menu**. In regular JavaScript, we can use **arrays** for creating lists. In React, rendering lists efficiently is critical for maintaining performance and ensuring a smooth user experience. React provides powerful tools to handle lists, allowing developers to create dynamic

and responsive applications.

In this tutorial, we will learn about **React lists with examples**. We will cover React list fundamentals like **creating lists**, **traversing lists**, and **rendering lists** in React with examples. Let's start by learning how to create and traverse React lists.

## Table of Content

- [Steps to Create and Traverse React JS Lists](#)
- [Rendering lists inside Components](#)
- [Key in React List](#)

## Rendering Lists with the `map()` Function

We can create lists in React just like we do in regular **JavaScript** i.e. by storing the list in an **array**. To traverse a list we will use the `map()` function. To create a React list, follow these given steps:

**Step 1:** Create a list of elements in React in the form of an array and store it in a variable. We will render this list as an unordered list element in the browser.

**Step 2:** We will then traverse the list using the JavaScript `map()` function and update elements to be enclosed between `<li>` `</li>` elements.

**Step 3:** Finally we will wrap this new list within `<ul>` `</ul>` elements and render it to the DOM.

## React List Examples

**Example:** This example implements a simple list in ReactJS.

```
// Filename - index.js

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

const numbers = [1,2,3,4,5];

const updatedNums = numbers.map((number)=>{
  return <li>{number}</li>;
});

ReactDOM.render(
  <ul>
    {updatedNums}
  </ul>,
  document.getElementById('root')
);
```

**Output:** The above code will render an unordered list as shown below

- 1
- 2
- 3
- 4
- 5

ReactJS Lists

## Rendering lists inside Components

In the above code in React, we directly rendered the list to the **DOM**. But usually, this is not a good practice to render lists in React. We already have talked about the uses of **Components** and have seen that everything in

React is built as individual components.

Consider the example of a Navigation Menu. It is obvious that in any website the items in a navigation menu are not hard coded. This item is fetched from the database and then displayed as a list in the browser. So from the component's point of view, we can say that we will pass a list to a component using **props** and then use this component to render the list to the DOM.

We can update the above code in which we have directly rendered the list to now a component that will accept an array as props and return an unordered list.

```
// Filename - index.js

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

// Component that will return an
// unordered list
function Navmenu(props)
{
    const list = props.menuitems;

    const updatedList = list.map((listItems)=>{
        return <li>{listItems}</li>;
    });

    return(
        <ul>{updatedList}</ul>
    );
}

const menuItems = [1,2,3,4,5];

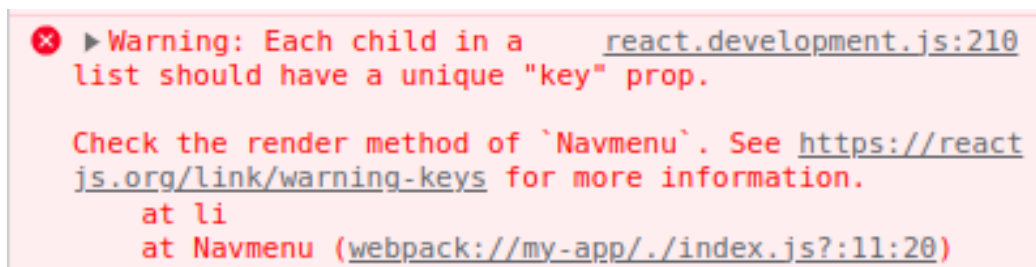
ReactDOM.render(
    <Navmenu menuitems = {menuItems} />,
    document.getElementById('root')
);
```

## Output:

- 1
- 2
- 3
- 4
- 5

## ReactJS Lists

**The above code will give a warning in the console of the browser like:**



## ReactJS Lists Warning

The above warning message says that each of the list items in our unordered list should have a unique key.

## Using Keys in Lists

**Keys** are an important aspect of rendering lists in React. They help React identify which items have changed, are added, or are removed. Providing a unique key for each list item significantly improves performance and avoids potential bugs.

**Example:** This is the updated code for React List with keys:

```

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

// Component that will return an
// unordered list
function Navmenu(props)
{
    const list = props.menuitems;

    const updatedList = list.map((listItems)=>{
        return(
            <li key={listItems.toString()}>
                {listItems}
            </li>
        );
    });

    return(
        <ul>{updatedList}</ul>
    );
}

const menuItems = [1,2,3,4,5];

ReactDOM.render(
    <Navmenu menuitems = {menuItems} />,
    document.getElementById('root')
);

```

**Output:** In the below-shown output you can see the rendered output is the same but this time without any warning in the console.

- 1
- 2
- 3
- 4
- 5

ReactJS Lists with Keys

## Conclusion



In summary, React lists are arrays with values. To render the elements of an array we iterate over each element and create a JSX element for each item. We use keys to label list elements, and when the changes are made to the list we don't need to re-render the entire list.

This tutorial explains React lists and uses JavaScript codes to show examples of all different concepts. After completing this tutorial, you will be able to use lists in your React project.

## React Forms

**React Forms** are the components **used to collect and manage the user inputs**. These components includes the input elements like text field, check box, date input, dropdowns etc.

In HTML forms the data is usually handled by the DOM itself but in the case of React Forms data is handled by the react components.

### Table of Content

- React Forms
- Controlled Components
- Adding Forms in React
- Handling React Forms
- Submitting React Forms

- Multiple Input Fields
- Alternatives to Controlled Components

## React Forms

**React forms** are the way to collect the user data in a React application. React typically utilize controlled components to manage form state and handle user input changes efficiently. It provides additional functionality such as preventing the default behavior of the form which refreshes the browser after the form is submitted.

In React Forms, all the form data is stored in the React's component state, so it can handle the form submission and retrieve data that the user entered. To do this we use controlled components.

### Syntax:

```
<form action={handleSubmit}>
  <label>
    User name:
    <input type="text" name="username" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

## Controlled Components

In simple HTML elements like input tags, the value of the input field is changed whenever the user type. But, In React, whatever the value the user types we save it in state and pass the same value to the input tag as its value, so here DOM does not change its value, it is

controlled by react state. These are known as **Controlled Components**.

## Adding Forms in React

Forms in React can be easily added as a simple react element. Here are some examples.

**Example:** This example displays the text input value on the console window when the React onChange event triggers.

```
// Filename - src/index.js:

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

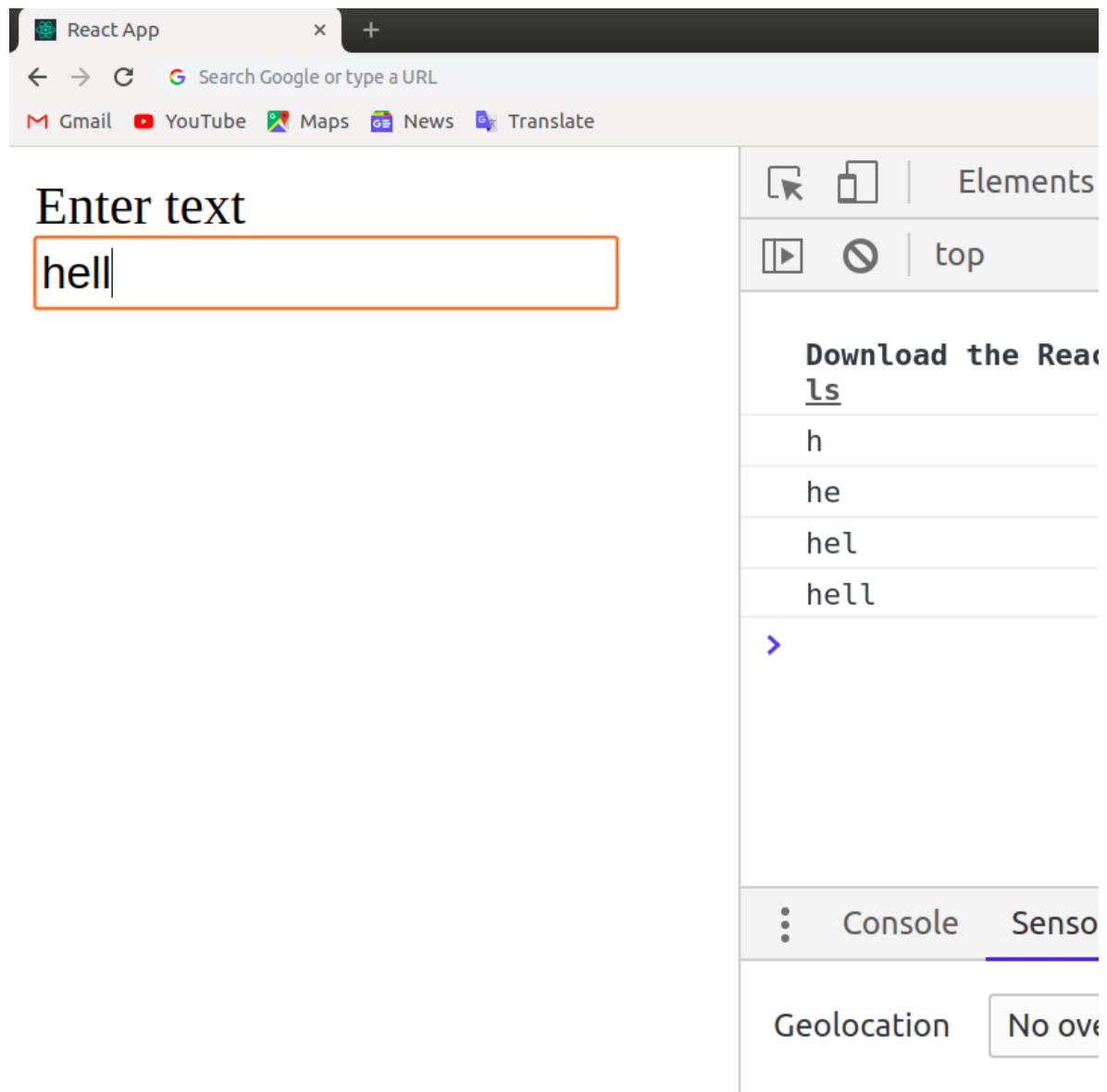
class App extends React.Component {

  onInputChange(event) {
    console.log(event.target.value);
  }

  render() {
    return (
      <div>
        <form>
          <label>Enter text</label>
          <input type="text"
onChange={this.onInputChange}/>
        </form>
      </div>
    );
  }
}

ReactDOM.render(<App />,
  document.querySelector('#root'));
```

## Output:



## Handling React Forms

In HTML the HTML DOM handles the input data but in react the values are stored in state variable and form data is handled by the components.

**Example:** This example shows updating the value of `inputValue` each time user changes the value in the input field by calling the `setState()` function.

```
// Filename - index.js

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

class App extends React.Component {
  state = { inputValue: '' };

  render() {
    return (
      <div>
        <form>
          <label> Enter text </label>
          <input type="text"
            value={this.state.inputValue}
            onChange={(e) =>
this.setState(
                      { inputValue:
e.target.value }}} />
          </form>
          <br />
          <div>
            Entered Value:
            {this.state.inputValue}
          </div>
        </div>
      );
    }
  }

ReactDOM.render(<App />,
  document.querySelector('#root'));
```

**Output:**

---

Enter text

Entered Value: hello

## Submitting React Forms

The submit action in react form is done by using the event handler `onSubmit` which accepts the submit function.

**Example:** Here we just added the React onSubmit event handler which calls the function `onFormSubmit` and it prevents the browser from submitting the form and reloading the page and changing the input and output value to 'Hello World!'.

```
// Filename - index.js
```

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

class App extends React.Component {
  state = { inputValue: "" };
}
```

```

    onFormSubmit = (event) => {
        event.preventDefault();
        this.setState({ inputValue: "Hello
World!" });
    };

    render() {
        return (
            <div>
                <form onSubmit={this.onFormSubmit}>
                    <label> Enter text </label>
                    <input
                        type="text"
                        value={this.state.inputValue}
                        onChange={(e) =>
                            this.setState({
                                inputValue:
e.target.value,
                                })
                        }
                    />
                </form>
                <br />
                <div>
                    Entered Value:
                    {this.state.inputValue}
                </div>
            </div>
        );
    }
}

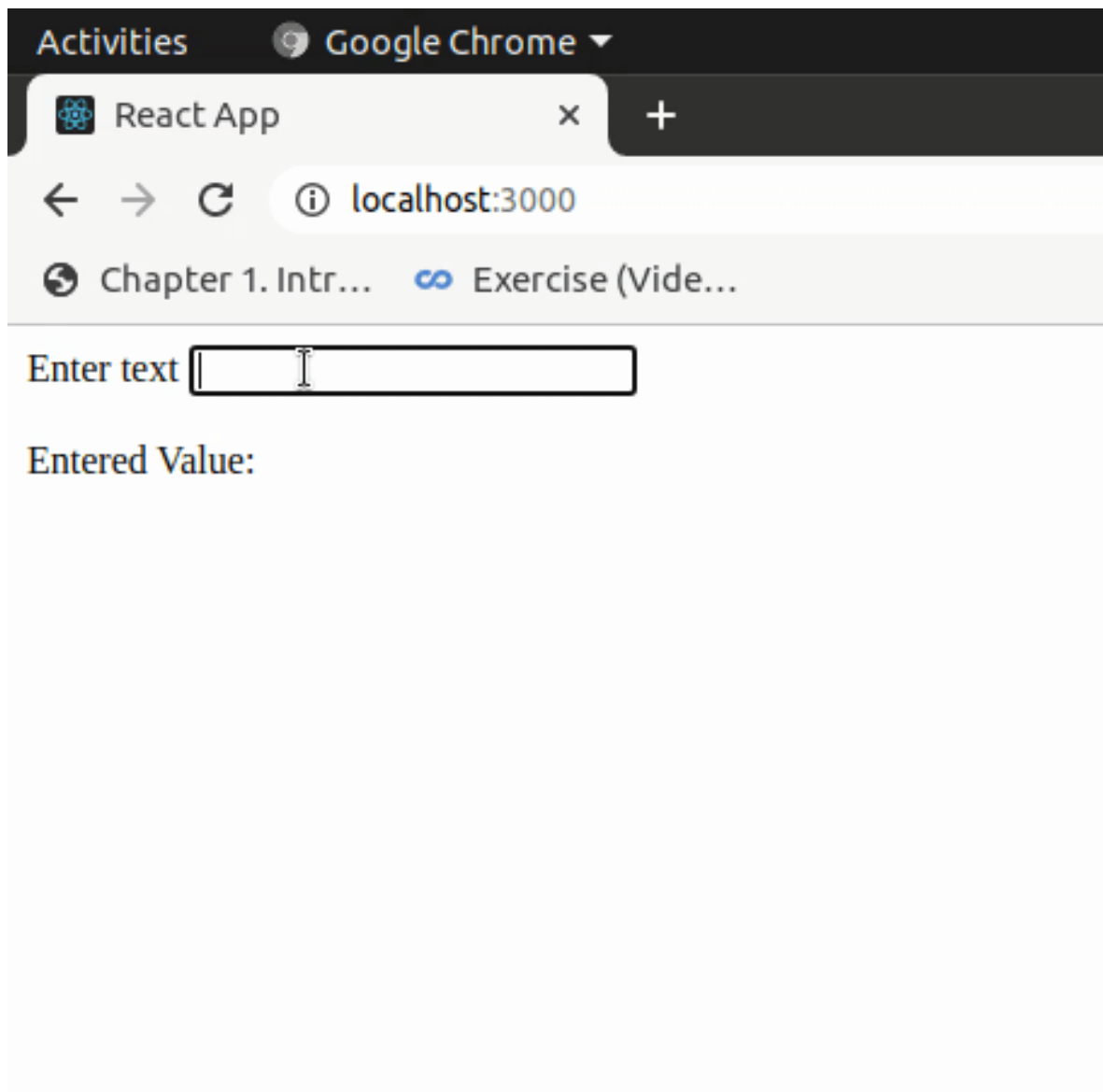
```

```

ReactDOM.render(<App />,
document.querySelector("#root"));

```

**Output:**



## Multiple Input Fields

React Forms allow to handle multiple inputs in a single form. Other types of input fields present in Forms are

- Textarea tag
- Select tag

### Textarea

The textarea tag defines the element by its children i.e., enclosed in the tags. In React we use the value prop instead.



## Syntax:

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

Here,

- **text:** refers to the state variable in which the value is stored
- **handleChange:** is the function to be executed to update the state when the onChange event triggers.

## Select

The select tag defines the element by its children i.e., enclosed in the tags. In React similar to text.area we use the value prop instead.

## Syntax:

```
<select value={this.state.value}
  onChange={this.handleChange}>
  <option value="HTML">HTML</option>
  <option value="CSS">CSS</option>
  <option value="JavaScript">JavaScript</option>
  <option value="React">React</option>
</select>
```

Here,

- **this.state.value:** refers to the state variable in which the value is stored
- **this.handleChange:** is the function to be executed to update the state when the onChange event triggers.

## Multiple Input Fields

**Example:** This example demonstrate handling multiple input fields in a single Form component.

```

import React from "react";
// import ReactDOM from 'react-dom/client';
import "./index.css";
// import App from './App';
// import reportWebVitals from './reportWebVitals';

// Filename - index.js

import ReactDOM from "react-dom";

class App extends React.Component {
  state = { username: "", email: "" };

  onFormSubmit = (event) => {
    event.preventDefault();
    this.setState({
      username: "hello123",
      email: "abc@hello.org",
    });
  };

  render() {
    return (
      <div
        style={{
          margin: "auto",
          marginTop: "20px",
          textAlign: "center",
        }}
      >
        <form onSubmit={this.onFormSubmit}>
          <label> Enter username: </label>
          <input
            type="text"
            value={this.state.username}
            onChange={(e) =>
              this.setState((prev) =>
                ({
                  ...prev,
                  username:
e.target.value,
                })
              )
            }
          />
        </form>
      </div>
    );
  }
}

```

```

        />
        <br />
        <br />
        <label>Enter Email Id:</label>
        <input
            type="email"
            value={this.state.email}
            onChange={(e) =>
                this.setState((prev) =>
                    ({
                        ...prev,
                        email:
e.target.value,
                    })))
        ></input>
        <br />
        <br />
        <input type="submit"
value={"Submit"} />
    </form>
    <br />
    <div>
        Entered Value:
        {this.state.username}
    </div>
    </div>
    );
}
}
const root = ReactDOM.createRoot(
    document.getElementById("root")
);
root.render(
    <React.StrictMode>
        <App />
    </React.StrictMode>
);

```

**Output:**

Enter username:

Enter Email Id:

Entered Value: gfg123

## Alternatives to Controlled Components

Controlled components are commonly used to manage form state by binding form elements to React state. The alternative to controlled components is uncontrolled components. In uncontrolled components, instead of managing the form data through React state, you let the DOM handle the form elements and directly interact with them. To know more difference check this article on [Controlled vs Uncontrolled Components in ReactJS](#).

## ReactJS Keys

React JS keys are a way of providing a unique identity to each item while creating the React JS Lists so that React can identify the element to be processed.

Table of Content

- [What is a key in React ?](#)

- Assigning keys to the list
- Issue with index as keys
- Difference between keys and props in React
- Using Keys with Components
- Incorrect usage of keys
- Correct usage of keys
- Uniqueness of Keys

## What is a key in React ?

A “key” is a special string attribute you need to include when creating lists of elements in React. Keys are used in React to identify which items in the list are changed, updated, or deleted.

Keys are used to give an identity to the elements in the lists. It is recommended to use a string as a key that uniquely identifies the items in the list.

## Assigning keys to the list

You can assign the array indexes as keys to the list items. The below example assigns array indexes as keys to the elements.

### Syntax:

```
const numbers = [1, 2, 3, 4, 5];
const updatedNums = numbers.map((number,
index) =>
    <li key={index}>
        {number}
    </li>
);
```

## Issue with index as keys

Assigning indexes as keys is **highly discouraged** because if the elements of the arrays get reordered in the future then it will get confusing for the developer as the keys for the elements will also change.

## **Difference between keys and props in React**

Keys are not the same as props, only the method of assigning “key” to a component is the same as that of props. Keys are internal to React and can not be accessed from inside of the component like props. Therefore, we can use the same value we have assigned to the Key for any other prop we are passing to the Component.

## **Using Keys with Components :**

Consider a situation where you have created a separate component for list items and you are extracting list items from that component. In that case, you will have to assign keys to the component you are returning from the iterator and not to the list items. That is you should assign keys to `<Component />` and not to `<li>` A good practice to avoid mistakes is to keep in mind that anything you are returning from inside of the `map()` function is needed to be assigned a key.

## **Incorrect usage of keys:**

**Example 1:** The below code shows the **incorrect usage** of keys:

```

import React from "react";
import ReactDOM from "react-dom/client";
// Component to be extracted
function MenuItems(props) {
    const item = props.item;
    return <li key={item.toString()}>{item}</li>;
}

// Component that will return an
// unordered list
function Navmenu(props) {
    const list = props.menuitems;
    const updatedList = list.map((listItems) => {
        return <MenuItems item={listItems} />;
    });

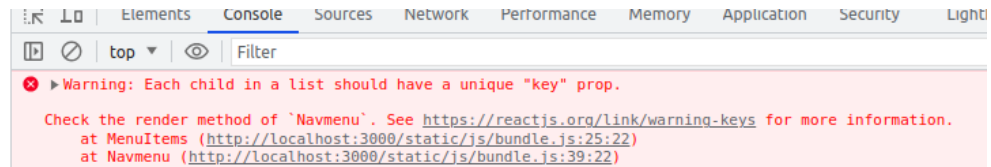
    return <ul>{updatedList}</ul>;
}

const menuItems = [1, 2, 3, 4, 5];
const root =
ReactDOM.createRoot(document.getElementById('root'))
);
root.render(
    <React.StrictMode>
        <Navmenu menuitems={menuItems} />
    </React.StrictMode>
);

```

**Output:** You can see in the below output that the list is rendered successfully but a warning is thrown to the console that the elements inside the iterator are not assigned keys. This is because we had not assigned the key to the elements we are returning to the map() iterator.

- 1
- 2
- 3
- 4
- 5



## Correct usage of keys:

**Example 2:** The below example shows the **correct** usage of keys.

```
import React from "react";
import ReactDOM from "react-dom";
// Component to be extracted
function MenuItems(props) {
  const item = props.item;
  return <li>{item}</li>;
}

// Component that will return an
// unordered list
function Navmenu(props) {
  const list = props.menuitems;
  const updatedList = list.map((listItems) => {
    return <MenuItems
key={listItems.toString()} item={listItems} />;
  });

  return <ul>{updatedList}</ul>;
}

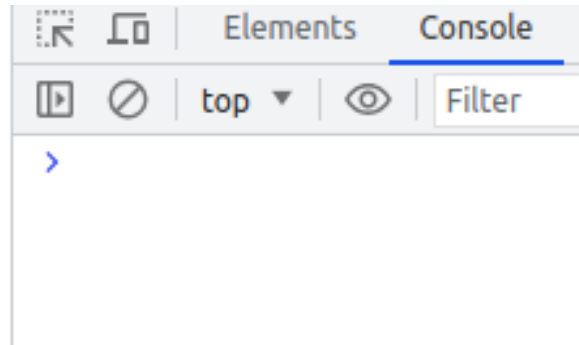
const menuItems = [1, 2, 3, 4, 5];

ReactDOM.render(
  <Navmenu menuitems={menuItems} />,
  document.getElementById("root")
);
```



**Output:** In the below-given output you can clearly see that this time the output is rendered but without any type of warning in the console, it is the correct way to use React Keys.

- 1
- 2
- 3
- 4
- 5



## Uniqueness of Keys:

We have told many times while discussing keys that keys assigned to the array elements must be unique. By this, we did not mean that the keys should be globally unique. All the elements in a particular array should have unique keys. That is, two different arrays can have the same set of keys.

**Example 3:** In the below code, we have created two different arrays `menuItems1` and `menuItems2`. You can see in the below code that the keys for the first 5 items for both arrays are the same still the code runs successfully without any warning.

```

import React from "react";
import ReactDOM from "react-dom";
// Component to be extracted
function MenuItems(props) {
  const item = props.item;
  return <li>{item}</li>;
}

// Component that will return an
// unordered list
function Navmenu(props) {
  const list = props.menuitems;
  const updatedList = list.map((listItems) => {
    return <MenuItems
key={listItems.toString()} item={listItems} />;
  });

  return <ul>{updatedList}</ul>;
}

const menuItems1 = [1, 2, 3, 4, 5];
const menuItems2 = [1, 2, 3, 4, 5, 6];

ReactDOM.render(
  <div>
    <Navmenu menuitems={menuItems1} />
    <Navmenu menuitems={menuItems2} />
  </div>,
  document.getElementById("root")
);

```

**Output:** This will be the output of the above code

- 1
- 2
- 3
- 4
- 5

- 1
- 2
- 3
- 4
- 5
- 6

## ReactJS Refs

**ReactJS Refs** are used to access and modify the **DOM elements** in the React Application. It creates a reference to the elements and uses it to modify them.

Table of Content

- What are React refs ?
- Creating refs in React
- Accessing Refs in React
- When to use refs
- When not to use refs

## What is Refs in React?

**Refs** are a function provided by React to access the

DOM element and the React elements created in components. They are used in cases where we want to change the value of a child component, without making use of props and state.

They allow us to interact with these elements outside the typical rendering workflow of React.

They have wide functionality as we can use callbacks with them.

## Creating refs in

### Example

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myCallRef = React.createRef();
  }
  render() {
    return <div ref={this.myCallRef} />;
  }
}
```

Detailed guide on [How to create refs in React JS?](#)

## Accessing Refs in React

In React, when a ref is passed to an element in render using the ref attribute, the underlying DOM element or React component becomes accessible at the current property of the ref.

```
const node = this.myCallRef.current;
```

Now, we are going to see how we can use refs in our code which will help you to understand the use case of refs better.

## Example

In this example, we use the target value of event e, for getting the value.

```
// Filename - App.js

// without refs
class App extends React.Component {
  constructor() {
    super();
    this.state = { sayings: "" };
  }
  update(e) {
    this.setState({ sayings: e.target.value });
  }
  render() {
    return (
      <div>
        Mukul Says{" "}
        <input
          type="text"
          onChange={this.update.bind(this)}
        />
        <br />
        <em>{this.state.sayings}</em>
      </div>
    );
  }
}

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

**Output:**

## Refs Current Properties

The current property value of refs depends on the **ref target**. Look at the table below, to understand the difference.

Target Type		current Property Value
HTML element	DOM element object	
Custom React component (class component)	React component instance	

## More Examples of Refs in React

Let's look at some of the React code examples of refs. The examples will provide a better learning experience for master ReactJS refs.

**Example 1:** In this example, we use refs to add a callback function indirectly with the help of the **update function** and **onChange event handler**.

```
// using refs
class App extends React.Component {
  constructor() {
    super();
    this.state = { sayings: "" };
  }
  update(e) {
    this.setState({ sayings:
this.refs.anything.value });
  }
}
```

```

    render() {
      return (
        <div>
          Mukul Says <input type="text"
ref="anything"
onChange={this.update.bind(this)} />
          <br />
          <em>{this.state.sayings}</em>
        </div>
      );
    }
  }
ReactDOM.render(< App />,
document.getElementById('root'));

```

### Output:

---

Mukul Says   
*using ref :)*

**Example 2:** In this example, we directly define callback function within ref.

```

// Filename - App.js

// callback used inside ref
class App extends React.Component {
  constructor() {
    super();
    this.state = { sayings: "" };
  }
  update(e) {
    this.setState({ sayings: this.a.value });
  }
  render() {
    return (
      <div>
        Mukul Says{" "}
        <input
          type="text"

```

```

        ref={({call_back) => {
            this.a = call_back;
        }}
        onChange={this.update.bind(this)}
    />
    <br />
    <em>{this.state.sayings}</em>
</div>
    );
}
}
ReactDOM.render(<App />,
document.getElementById("root"));

```

### Output:

Mukul Says   
*callbacks too :)*

## When to use refs

Using refs provides a lot of benefits, and improves your web development experience. It is helpful in:

- Helpful when using third-party libraries.
- Helpful in animations.
- Helpful in managing focus, media playback, and text selection.

## When not to use refs

- Should not be used with functional components because they don't have instances.
- Not to be used on things that can be done declaratively.
- When using a library or framework that provides its methods for managing such as Redux or MobX.

## Conclusion

React refs are useful to interact with the DOM structure of components. They can directly access and



manipulate the DOM elements. This guide teaches the purpose of refs in React, how to create refs, and how to use refs in React with examples.

# ReactJS Rendering Elements

## What are React Elements?

React elements are different from DOM elements as React elements are simple JavaScript objects and are efficient to create. React elements are the building blocks of any React app and should not be confused with React components which will be discussed in further articles.

## Rendering an Element in React

In order to render any element into the Browser DOM, we need to have a container or root DOM element. It is almost a convention to have a div element with the id="root" or id="app" to be used as the root DOM element. Let's suppose our index.html file has the following statement inside it.

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

Now, in order to render a simple React Element to the root node, we must write the following in the **App.js** file.

**javascript**

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

class App extends Component {

  render() {
    return (
      <div>
        <h1>Welcome to hello!</h1>
      </div>

    );
  }
}

export default App;
```

## Updating an Element in React

React Elements are immutable i.e. once an element is created it is impossible to update its children or attribute. Thus, in order to update an element, we must use the `render()` method several times to update the value over time. Let's see this as an example.

Write the following code in **App.js** file of your project directory.

**javascript**

```

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

function App() {
  const myElement = (
    <div>
      <h1>Welcome to hello!</h1>
      <h2>{new Date().toLocaleTimeString()}</
h2>
    </div>
  );

  ReactDOM.render(
    myElement,
    document.getElementById("root")
  );
}

setInterval(App, 1000);

export default App;

```

In the above example, we have created a function `showTime()` that displays the current time, and we have set an interval of 1000ms or 1 sec that recalls the function each second thus updating the time in each call. For simplicity, we have only shown the timespan of one second in the given image.

## React Render Efficiency

React is chosen over the legacy of DOM updates because of its increased efficiency. React achieves this efficiency by using the virtual DOM and efficient differentiating algorithm.

In the example of displaying the current time, at each second we call the render method, and the virtual DOM

gets updated and then the differentiator checks for the particular differences in Browser DOM and the Virtual DOM and then updates only what is required such as in the given example the time is the only thing that is getting changed each time not the title “Welcome to hello!” thus React only updates the time itself making it much more efficient than conventional DOM manipulation.

### **Important Points to Note:**

- Calling the render() method multiple times may serve our purpose for this example, but in general, it is never used instead a stateful component is used which we will cover in further articles.
- A React Element is almost never used isolated, we can use elements as the building blocks of creating a component in React. Components will also be discussed in upcoming articles.

## **React Conditional Rendering**

React allows us to conditionally render components which means that the developer can decide which component to render on the screen on the basis of some predefined conditions. This is known as **conditional rendering**.

### **Table of Content**

- [Implementing Conditional Rendering](#)
- [1. Conditional Rendering using if-else Statement](#)

- 2. Conditional Rendering using logical && operator
- 3. Conditional Rendering using ternary operator
- Preventing Component from Rendering

## Implementing Conditional Rendering

There may arise a situation when we want to render something based on some condition. For example, consider the situation of handling a login/logout button. Both the buttons have different functions so they will be separate components. Now, the task is if a user is logged in then we will have to render the Logout component to display the logout button and if the user is not logged in then we will have to render the Login component to display the login button.

**Conditional rendering in React can be implemented in three ways:**

- Using if else Statement
- Using Logical && Operator
- Using ternary operator

## 1. Conditional Rendering using if-else Statement

We can create two components and create a boolean variable which decides the element to be rendered on the screen.

**if-else Conditional Rendering Syntax:**

```
function MainComponent(props) {
```

```

    const myBool = props.myBool;
    if (myBool) {
        return <Component1/>;
    } else{
        return <Component2/>;
    }
}

```

## if-else Statement Example:

Open your react project directory and edit the **Index.js** file from src folder:

```

import React from "react";
import ReactDOM from "react-dom";

// Message Component
// Message Component
function Message(props)
{
    if (props.isLoggedIn)
        return <h1>Welcome User</h1>;
    else
        return <h1>Please Login</h1>;
}

// Login Component
function Login(props) {
    return <button onClick={props.clickFunc}
>Login</button>;
}

// Logout Component
function Logout(props) {
    return <button onClick={props.clickFunc}
>Logout</button>;
}

// Parent Homepage Component
class Homepage extends React.Component {

```

```

    constructor(props) {
        super(props);

        this.state = { isLoggedIn: false };

        this.ifLoginClicked =
this.ifLoginClicked.bind(this);
        this.ifLogoutClicked =
this.ifLogoutClicked.bind(this);
    }

    ifLoginClicked() {
        this.setState({ isLoggedIn: true });
    }

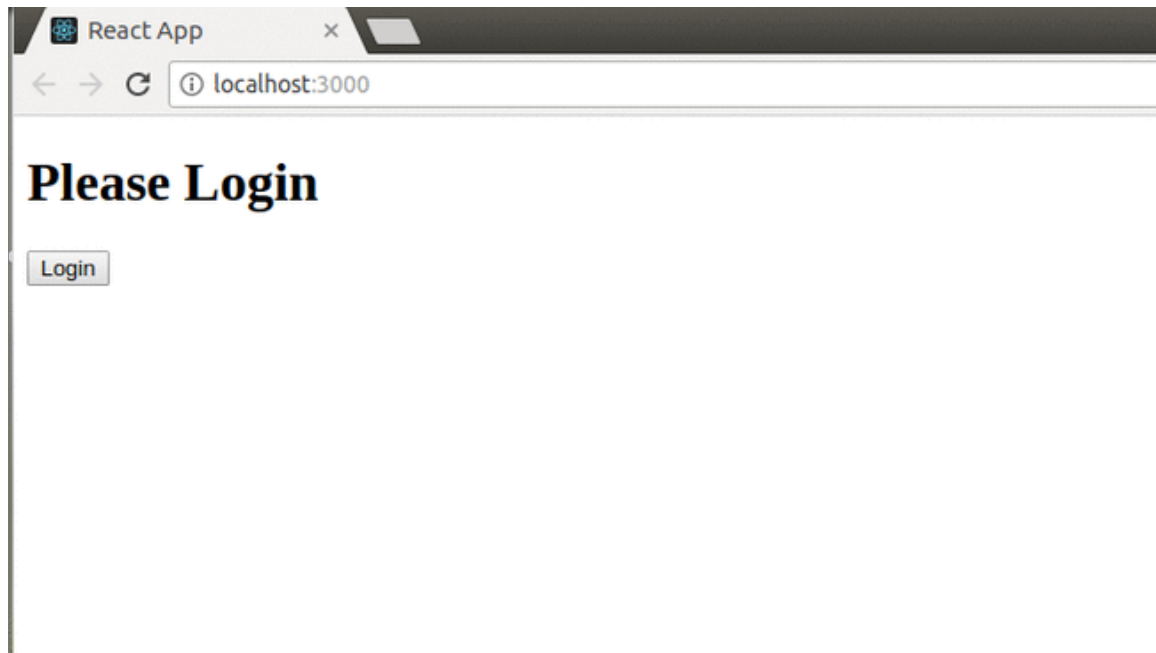
    ifLogoutClicked() {
        this.setState({ isLoggedIn: false });
    }

    render() {
        return (
            <div>
                <Message
                    isLoggedIn={ this.state.isLoggedIn } />
                { this.state.isLoggedIn ? (
                    <Logout
                        clickFunc={ this.ifLogoutClicked } />
                    ) : (
                        <Login
                            clickFunc={ this.ifLoginClicked } />
                    ) }
            </div>
        );
    }
}

ReactDOM.render(<Homepage />,
    document.getElementById("root"));

```

**Output:**



### Explanation:

In the above output, you can see that on clicking the Login button the message and button get's changed and vice versa.

## 2. Conditional Rendering using logical && operator

We can use the logical && operator along with some condition to decide what will appear in output based on whether the condition evaluates to true or false. Below is the syntax of using the logical && operator with conditions:

### logical && operator Syntax:

```
{  
  condition &&  
  
  // This section will contain
```



```
    // elements you want to return
    // that will be a part of output
}
```

If the **condition** provided in the above syntax evaluates to True then the elements right after the && operator will be a part of the output and if the condition evaluates to false then the code within the curly braces will not appear in the output.

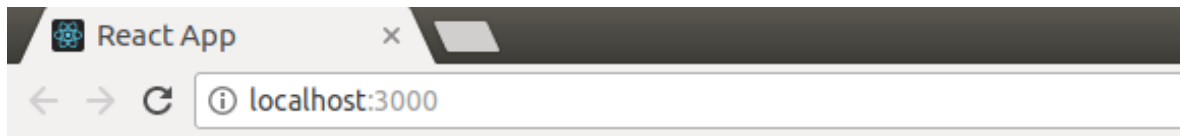
## logical && operator Example:

Below example is used to illustrate the above mentioned approach Open your react project directory and edit the **Index.js** file from src folder:

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

// Example Component
function Example() {
  const counter = 5;
  return (<div>
    {
      (counter==5) &&
      <h1>Hello World!</h1>
    }
    </div>
  );
}
ReactDOM.render(
  <Example />,
  document.getElementById('root')
);
```

**Output:**



# Hello World!

## Explanation:

You can clearly see in the above output that as the condition (`counter == 5`) evaluates to true so the `<h1>` element is successfully rendered on the screen.

## 3. Conditional Rendering using ternary operator

In JavaScript we have a short syntax for writing the if-else conditions due to which the code becomes shorter and easy to read. If the boolean returns true then the element on left will be rendered otherwise element on the right will be rendered

### Ternary Operator Syntax:

```
function MainComponent(props) {  
  const myBool = props.myBool;  
  return(  
    myBool ? <h1>Hello World! : <h2>Hello World!
```

```

        <>
            {myBool? <Component 1/>:
<Component 2/>}
        </>
    )
}

```

## Ternary Operator Example:

We will modify the first approach and render the elements using ternary operator

```

import React from "react";
import ReactDOM from "react-dom";

// Message Component
function Message(props) {
    return props.isLoggedIn ? <h1>Welcome User</h1> : <h1>Please Login</h1>;
}

// Login Component
function Login(props) {
    return <button onClick={props.clickFunc}>Login</button>;
}

// Logout Component
function Logout(props) {
    return <button onClick={props.clickFunc}>Logout</button>;
}

// Parent Homepage Component
class Homepage extends React.Component {
    constructor(props) {
        super(props);

        this.state = { isLoggedIn: false };
    }
}

```

```

        this.ifLoginClicked =
this.ifLoginClicked.bind(this);
        this.ifLogoutClicked =
this.ifLogoutClicked.bind(this);
    }

    ifLoginClicked() {
        this.setState({ isLoggedIn: true });
    }

    ifLogoutClicked() {
        this.setState({ isLoggedIn: false });
    }

    render() {
        return (
            <div>
                <Message
isLoggedIn={ this.state.isLoggedIn } />
                { this.state.isLoggedIn ? (
                    <Logout
clickFunc={ this.ifLogoutClicked } />
                ) : (
                    <Login
clickFunc={ this.ifLoginClicked } />
                ) }
            </div>
        );
    }
}

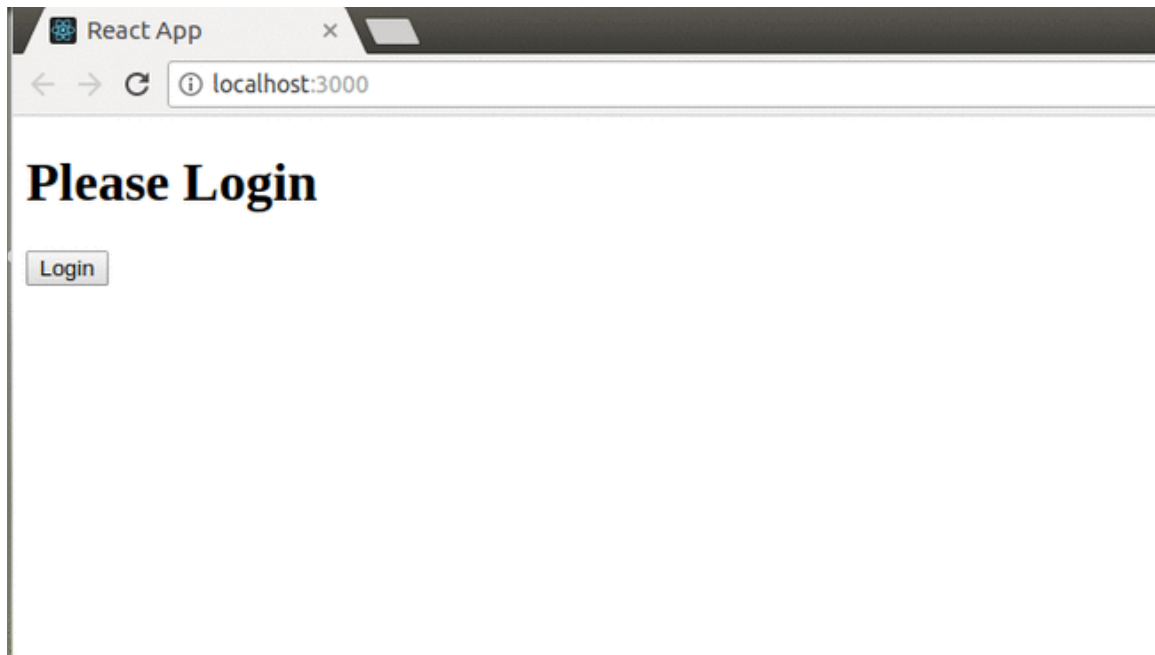
```

```

ReactDOM.render(<Homepage />,
document.getElementById("root"));

```

**Output:**



## Preventing Component from Rendering

It might happen sometimes that we may not want some components to render. To prevent a component from rendering we will have to return null as its rendering output. Consider the below example:

### Rendering Prevention Example:

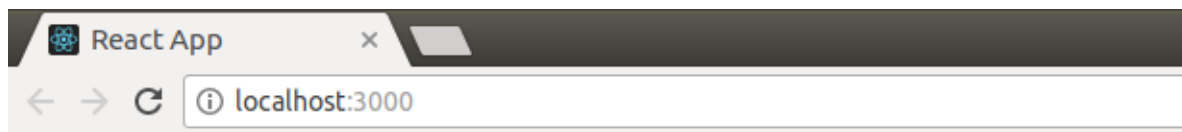
Open your react project directory and edit the **Index.js** file from src folder:

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

// Example Component
function Example(props)
{
    if(!props.toDisplay)
        return null;
    else
        return <h1>Component is rendered</h1>;
}

ReactDOM.render(
    <div>
        <Example toDisplay = {true} />
        <Example toDisplay = {false} />
    </div>,
    document.getElementById('root')
);
```

## Output:



**Component is rendered**

## Explanation:

You can clearly see in the above output that the Example component is rendered twice but the <h1>

element is rendered only once as on the second render of the Example component, null is returned as its rendering output.

## React Components

**Components in React** serve as independent and reusable code blocks for UI elements. They represent different parts of a web page and contain both **structure** and **behavior**. They are similar to **JavaScript functions** and make creating and managing complex user interfaces easier by breaking them down into smaller, reusable pieces.

### What are React Components?

**React Components** are the building block of React Application. They are the reusable code blocks containing logics and UI elements. They have the same purpose as **JavaScript functions** and return **HTML**. Components make the task of building UI much easier.

A UI is broken down into multiple individual pieces called components. You can work on components independently and then merge them all into a **parent component** which will be your final UI.

Components promote **efficiency** and **scalability** in web development by allowing developers to compose, combine, and customize them as needed.

You can see in the below image we have broken down the UI of hello's homepage into individual components.

Components in React return a piece of JSX code that tells what should be **rendered on the screen**.

## Types of Components in React

In React, we mainly have two types of components:

- **Functional Components**
- **Class based Components**

### Functional Component in React

**Functional components** are just like JavaScript functions that accept properties and return a React element.

We can create a functional component in React by writing a JavaScript function. These functions may or may not receive data as parameters, we will discuss this later in the tutorial. The below example shows a valid functional component in React:

#### **Syntax:**

```
function demoComponent() {  
    return (<h1>  
        Welcome Message!  
    </h1>);  
}
```

**Example:** Create a function component called welcome.



```
function welcome() {  
  return <h1>Hello, Welcome to hello!</h1>;  
}
```

## Class Component in React

The class components are a little more complex than the functional components. A class component can show **inheritance** and access data of other components.

Class Component must include the line “**extends React.Component**” to pass data from one class component to another class component. We can use JavaScript ES6 classes to create class-based components in React.

### Syntax:

```
class Democomponent extends React.Component {  
  render() {  
    return <h1>Welcome Message!</h1>;  
  }  
}
```

The below example shows a valid class-based component in React:

**Example:** Create a class component called welcome.

```
class Welcome extends Component {  
  render() {  
    return <h1>Hello, Welcome to hello!</h1>;  
  }  
}
```

## Functional Component vs Class

# Component

A functional component is best suited for cases where the component doesn't need to interact with other components or manage complex states. Functional components are ideal for **presenting static UI elements** or composing multiple simple components together under a single parent component.

While class-based components can achieve the same result, they are generally **less efficient** compared to functional components. Therefore, it's recommended to not use class components for general use.

## Rendering React Components

**Rendering Components** means turning your component code into the UI that users see on the screen.

React is capable of rendering user-defined components. To render a component in React we can initialize an element with a user-defined component and pass this element as the first parameter to **ReactDOM.render()** or directly pass the component as the first argument to the ReactDOM.render() method.

The below syntax shows how to initialize a component to an element:

```
const elementName = <ComponentName />;
```

In the above syntax, the ComponentName is the name of the user-defined component.

**Note:** The name of a component should always start

with a capital letter. This is done to differentiate a component tag from an HTML tag.

**Example:** This example renders a component named Welcome to the Screen.

```
// Filename - src/index.js:

import React from "react";
import ReactDOM from "react-dom";

// This is a functional component
const Welcome = () => {
  return <h1>Hello World!</h1>;
};

ReactDOM.render(
  <Welcome />,
  document.getElementById("root")
);
```

**Output:** This output will be visible on the **http://localhost:3000/** on the browser window.

# Hello World!

## Explanation:

Let us see step-wise what is happening in the above example:

- We call the ReactDOM.render() as the first parameter.
- React then calls the component Welcome, which returns <h1>Hello World!</h1>; as the result.
- Then the ReactDOM efficiently updates the DOM to match with the returned element and renders that

element to the DOM element with id as “root”.

## Props

- **Props(short for properties)** are a way to pass data from the parent component to the child component.
- Props are like **function arguments**, you can use them as attributes in components.

### Example:

```
function Message(props) {  
  return <h2>{props.text}</h2>;  
}
```

```
const root =  
ReactDOM.createRoot(document.getElementById('root'));  
root.render(<Message text="Hello, world!" />);
```

## Components in Components

We can call components inside another component

### Example:

```
// Filename - src/index.js:
```

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

```
const Greet = () => {  
  return <h1>Hello Geek</h1>  
}
```

```
// This is a functional component  
const Welcome = () => {  
  return <Greet />;  
};
```

```
ReactDOM.render(  

```

```
    <Welcome />,
    document.getElementById("root")
  );
```

The above code will give the same output as other examples but here we have called the Greet component inside the Welcome Component.

For more information on components open [Component Set 2](#)

## Conclusion

Components in React allow developers to divide the page UI into many small parts. These parts work independently and can be used multiple times just like functions.

## ReactJS I Components – Set 2

In our previous article on [ReactJS I Components](#) we had to discuss components, types of components, and how to render components. In this article, we will see some more properties of components.

**Composing Components:** Remember in our previous article, our first example of hello's homepage which we used to explain components? Let's recall what we have told, "we can merge all of these individual components to make a parent component". This is what we call composing components. We will now create individual components named Navbar, Sidebar, ArticleList and merge them to create a parent component named App and then render this App component.

The below code in the index.js file explains how to do this:

**Filename- App.js:**

**javascript**

```

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

// Navbar Component
const Navbar=()=>
{
    return <h1>This is Navbar.< /h1>
}

// Sidebar Component
const Sidebar=()=> {
    return <h1>This is Sidebar.</h1>
}

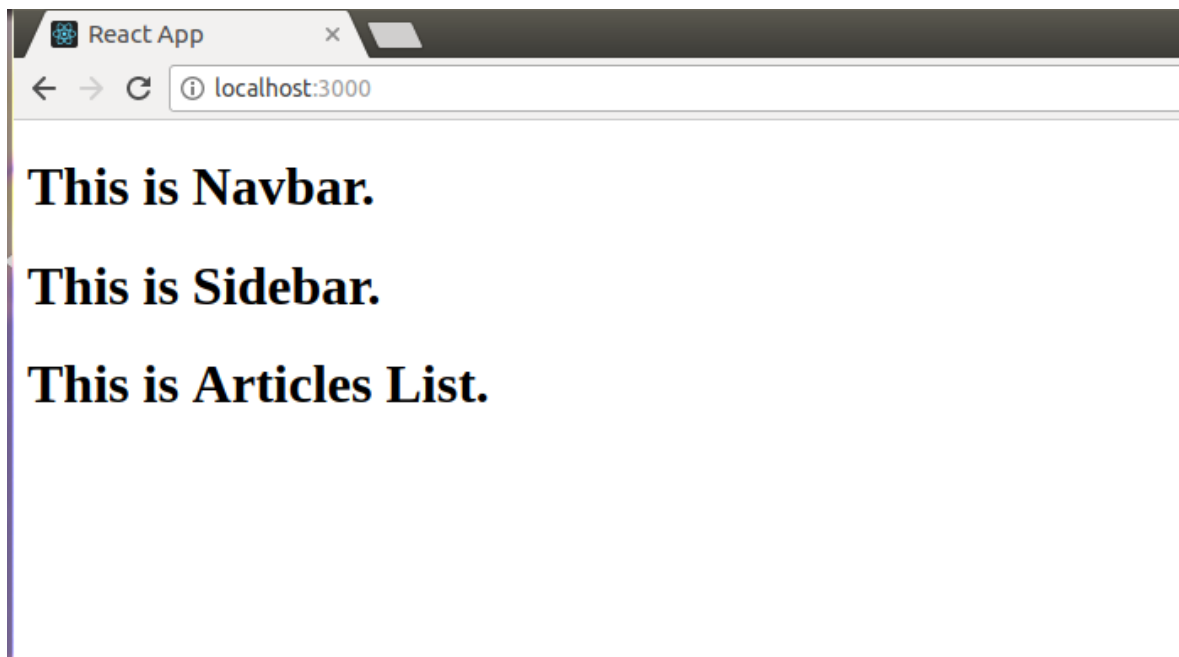
// Article list Component
const ArticleList=()=>
{
    return <h1>This is Articles List.</h1>
}

// App Component
const App=()=>
{
    return (
        <div>
            <Navbar />
            <Sidebar />
            <ArticleList />
        </div>
    );
}

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

```

**Output:**



You can see in the above output that everything worked well, and we managed to merge all the components into a single component App.

**Decomposing Components:** Decomposing a Component means breaking down the component into smaller components. We have told the thing about composing smaller components to build a parent component from the very start when we started discussing components repeatedly. Let us see why there is a need to do so. Suppose we want to make a component for an HTML form. Let's say our form will have two input fields and a submit button. We can create a form component as shown below:

**Filename- App.js:**

**javascript**

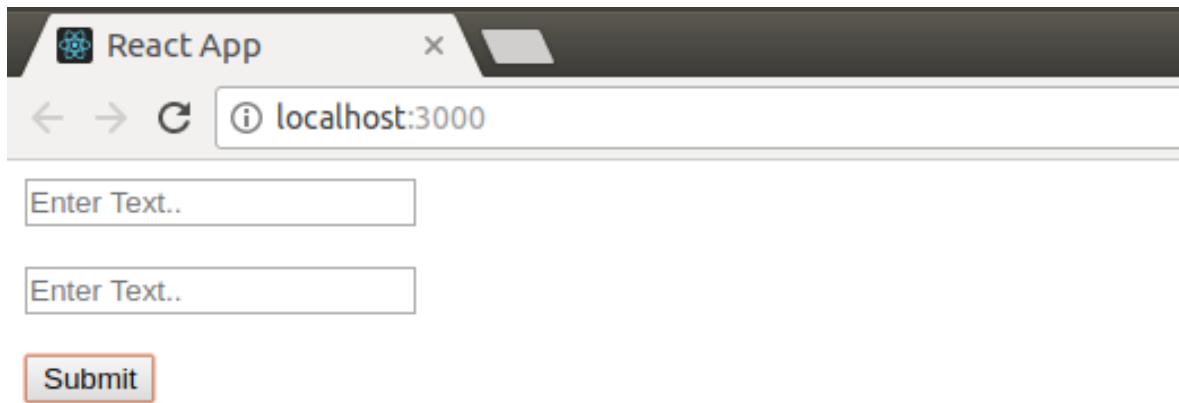


```
import React from 'react';
import ReactDOM from 'react-dom';

const Form=()=>>
{
    return (
        <div>
            <input type = "text" placeholder =
"Enter Text.." />
            <br />
            <br />
            <input type = "text" placeholder =
"Enter Text.." />
            <br />
            <br />
            <button type = "submit">Submit</button>
        </div>
    );
}

ReactDOM.render(
    <Form />,
    document.getElementById("root")
);
```

**Output:**



The screenshot shows a web browser window with a single tab titled "React App". The address bar displays "localhost:3000". The page content consists of two text input fields, each with the placeholder text "Enter Text..", and a "Submit" button below them.

The above code works well to create a form. But let us say now we need some other form with three input fields. To do this we will have to again write the complete code with three input fields now. But what if we have broken down the Form component into two smaller components, one for the input field and another one for the button? This could have increased our code reusability to a great extent. That is why it is recommended to React to break down a component into the smallest possible units and then merge them to create a parent component to increase the code modularity and reusability. In the below code the component Form is broken down into smaller components Input and Button.

**Filename- App.js:**

**javascript**

```

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

// Input field component
const Input=()=>>
{
    return(
        <div>
            <input type="text" placeholder="Enter
Text.." />
            <br />
            <br />
        </div>
    );
}

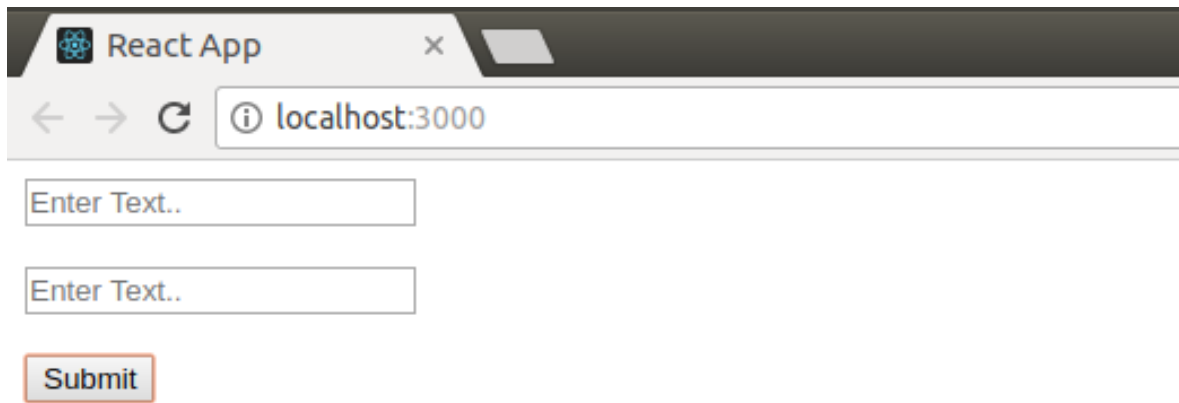
// Button Component
const Button=()=>>
{
    return <button type = "submit">Submit</
button>;
}

// Form component
const Form=()=>>
{
    return (
        <div>
            <Input />
            <Input />
            <Button />
        </div>
    );
}

ReactDOM.render(
    <Form />,
    document.getElementById("root")
);

```

**Output:**



# ReactJS Functional Components

Functional Component is one way to create components in a React Application. React.js Functional Components helps to create UI components in a Functional and more concise way. In this article, we will learn about functional components in React, different ways to call the functional component, and also learn how to create the functional components. We will also demonstrate the use of hooks in functional components

## Table of Content

- [Functional components in React :](#)
- [Ways to call the functional component:](#)
- [Problem with using functional components](#)
- [Advantage of using hooks in functional components](#)

# Functional Components in React :

**ReactJS Functional components** are some of the more common components that will come across while working in React. These are simply JavaScript functions. We can create a functional component in React by writing a JavaScript function. These functions may or may not receive data as parameters. In the functional Components, the return value is the JSX code to render to the DOM tree.

## Ways to call the functional component:

We can call the functions in JavaScript in other ways as follows:

### 1. Call the function by using the name of the function followed by the Parentheses.

```
// Example of Calling the function with
function name followed by Parentheses
function Parentheses() {
    return (<h1>
                We can call function using
name of the                function followed by
                Parentheses
                </h1>);
}
const root =
ReactDOM.createRoot(document.getElementById('r
oot'));
root.render(Parentheses());
```

### 2. Call the function by using the functional component method.

```
// Example of Calling the function using
component call
function Comp() {
    return (<h1> As usual we can call the
function using component call</h1>);
}
const root =
ReactDOM.createRoot(document.getElementById('r
oot'));
root.render(<Comp />);
```

Now, We will use the functional component method to create a program and see how functional components render the component in the browser.

## **Steps to create the React application:**

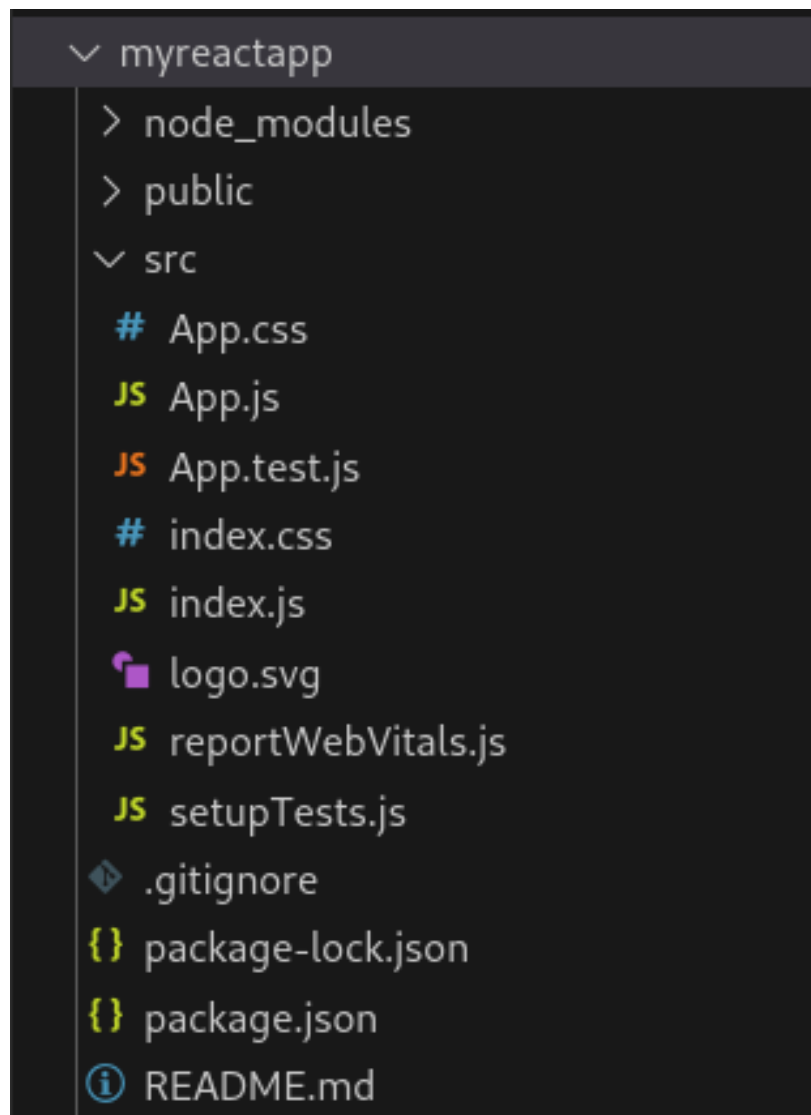
### **Step 1: Create React Project**

```
npm create-react-app myreactapp
```

**Step 2:** Change your directory and enter your main folder charting as

```
cd myreactapp
```

### **Project Structure:**



**Example 1:** This example demonstrates the creation of functional components.

## Javascript

```
// Filename - index.js

import React from "react";
import ReactDOM from "react-dom";
import Demo from "./App";

const root = ReactDOM.createRoot(
  document.getElementById("root")
);
root.render(
  <React.StrictMode>
    <Demo />
  </React.StrictMode>
);
```

## Javascript

```
//Filename - App.js

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

const Demo=()=>{return <h1>Welcome to hello</h1>};
export default Demo;
```

**Step to run the application:** Open the terminal and type the following command.

npm start

**Output:** Open the browser and our project is shown in the URL <http://localhost:3000/>

## Problem with using functional components

**Functional components** lack a significant amount of features as compared to **class-based components** and



they do not have access to dedicated state variables like **class-based components**.

## Advantage of using hooks in functional components

The problem discussed above is solved with the help of a special ReactJS concept called “hooks”. ReactJS has access to a special hook called **useState()**. The **useState()** is used to initialize only one state variable to multiple state variables. The first value returned is the initial value of the state variable, while the second value returned is a reference to the function that updates it.

**Example 2:** This example demonstrates the use of **useState()** hook in functional component.

## Javascript

```
// Filename - index.js

import React from "react";
import ReactDOM from "react-dom";
import Example from "../App";

const root = ReactDOM.createRoot(
  document.getElementById("root")
);
root.render(
  <React.StrictMode>
    <Example />
  </React.StrictMode>
);
```

## Javascript

```
// Filename - App.js

import React, { useState } from "react";

const Example = () => {
  const [change, setChange] = useState(true);
  return (
    <div>
      <button onClick={() => setChange(!
change)}>>
        Click Here!
      </button>
      {change ? (
        <h1>Welcome to hello</h1>
      ) : (
        <h1>A Computer Science Portal for
hello</h1>
      )}
    </div>
  );
};

export default Example;
```

**Step to run the application:** Open the terminal and type the following command.

npm start

**Output:** Open the browser and our project is shown in the URL <http://localhost:3000/>

Functional components do not have access to lifecycle functions like class-based components do since lifecycle functions need to be defined within the boundaries of a class. A special React hook called useEffect() needs to be used. It is worth noting that **useEffect()** isn't an exact duplicate of the lifecycle functions – it works and

behaves in a slightly different manner.

**Example 3:** This example demonstrates the use of `useEffect()` hook.

## Javascript

```
// Filename - index.js

import React from "react";
import ReactDOM from "react-dom";
import Example from "./App";

const root = ReactDOM.createRoot(
  document.getElementById("root")
);
root.render(
  <React.StrictMode>
    <Example />
  </React.StrictMode>
);
```

## Javascript

```
// Filename - App.js

import React, { useEffect } from "react";

const Example = () => {
  useEffect(() => {
    console.log("Mounting...");
  });
  return <h1>he...llo.!!</h1>;
};
export default Example;
```

**Step to run the application:** Open the terminal and type the following command.

```
npm start
```

**Output:** Open the browser and our project is shown in the URL <http://localhost:3000/>

Data is passed from the parent component to the child components in the form of props. ReactJS does not allow a component to modify its own props as a rule. The only way to modify the props is to change the props being passed to the child component. This is generally done by passing a reference of a function in the parent component to the child component.

**Example 4:** This example demonstrates the use of props.

## Javascript

```
// Filename - index.js

import React from "react";
import ReactDOM from "react-dom";
import PropsExample from "../App";

const root = ReactDOM.createRoot(
  document.getElementById("root")
);
root.render(
  <React.StrictMode>
    <PropsExample />
  </React.StrictMode>
);
```

# Javascript

```
// Filename - App.js

import React, { useState } from "react";

const Example = (props) => {
  return <h1>{props.data}</h1>;
};

const PropsExample = () => {
  // const [change, setChange] = useState(true);
  const [change, setChange] = useState(false);
  return (
    <div>
      <button onClick={() => setChange(!
change)}>
        Click Here!
      </button>
      {change ? (
        <Example data="Welcome to hello" />
      ) : (
        <Example data="A Computer Science
Portal for he"lo />
      )}
    </div>
  );
};

export default PropsExample;
```

**Step to run the application:** Open the terminal and type the following command.

npm start

**Output:** Open the browser and our project is shown in the URL <http://localhost:3000/>

## React Lifecycle

**React lifecycle** starts from its initialization and ends when it is unmounted from the DOM. There are several methods defined for different phases of the lifecycle of React Components.

## Table of Content

- [React Lifecycle](#)
- [Lifecycle of React Components](#)
- [Phases of Lifecycle in React Components](#)
- [Implementing the Component Lifecycle methods](#)

## React Lifecycle

**React Lifecycle** is defined as the series of methods that are invoked in different stages of the component's existence. The definition is pretty straightforward but what do we mean by different stages? A React Component can go through four stages of its life as follows.

## Lifecycle of React Components:

Each React Component go through the given Phases.

### 1. Initialization phase

This is the stage where the component is constructed with the given Props and default state. This is done in the constructor of a Component Class.

### 2. Mounting Phase

Mounting is the stage of rendering the JSX returned by the render method itself.

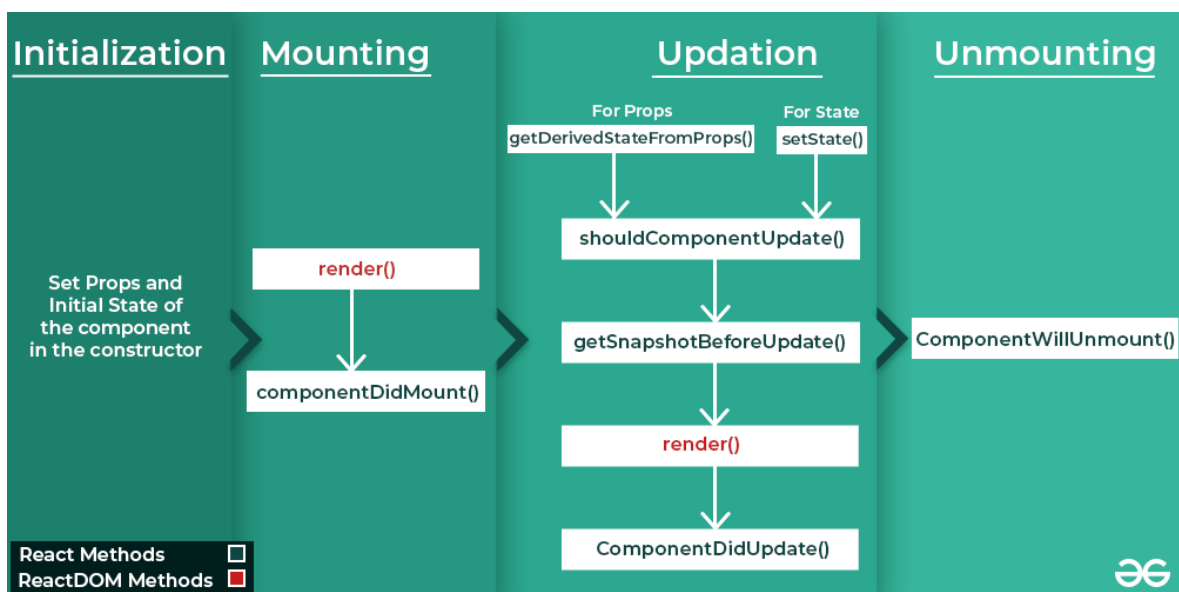
### 3. Updating

Updating is the stage when the state of a component is updated and the application is repainted.

### 4. Unmounting

As the name suggests Unmounting is the final step of the component lifecycle where the component is removed from the page.

React provides the developers with a set of predefined functions that if present are invoked around specific events in the lifetime of the component. Developers are supposed to override the functions with the desired logic to execute accordingly. We have illustrated the gist in the following diagram.



Now let us describe each phase and its corresponding

functions.

# Phases of Lifecycle in React Components

## 1. Initialization

In this phase, the developer has to define the props and initial state of the component this is generally done in the constructor of the component. The following code snippet describes the initialization process.

```
class Clock extends React.Component {
  constructor(props)
  {
    // Calling the constructor of
    // Parent Class React.Component
    super(props);

    // Setting the initial state
    this.state = { date : new Date() };
  }
}
```

## 2. Mounting

Mounting is the phase of the component lifecycle when the initialization of the component is completed and the component is mounted on the DOM and rendered for the first time on the webpage. Now React follows a default procedure in the Naming Conventions of these predefined functions where the functions containing “Will” represents before some specific phase and “Did” represents after the completion of that phase. The mounting phase consists of two such predefined



functions as described below.

- constructor
- static getDerivedStateProps
- render()
- componentDidMount()

constructor():

Method to initialize state and bind methods. Executed before the component is mounted.

constructor example:

```
// Filename - src/index.js:

import React from "react";
import ReactDOM from "react-dom/client";

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

  render() {
    return (
      <div>
        <h1>
          hello.org, Hello
          {this.state.hello}
        </h1>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(
  document.getElementById("root")
);
root.render(<Test />);
```

### static getDerivedStateFromProps

Used for updating the state based on props. Executed before every render.

Example:

```
// Filename - src/index.js:

import React from "react";
import ReactDOM from "react-dom/client";

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

  render() {
    return (
      <div>
        <h1>
          hello.org, Hello
          {this.state.hello}
        </h1>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(
  document.getElementById("root")
);
root.render(<Test greet="hello!"/>);
```

render() method:

Responsible for rendering JSX and updating the DOM.

**render() Example:**

```
// Filename - src/index.js:

import React from "react";
import ReactDOM from "react-dom/client";

class Test extends React.Component {
  render() {
    return (
      <div>
        <h1>
          hello
        </h1>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(
  document.getElementById("root")
);
root.render(<Test />);
```

### **componentDidMount() Function**

This function is invoked right after the component is mounted on the DOM i.e. this function gets invoked once after the render() function is executed for the first time

### **componentDidMount() Example:**

```
// Filename - src/index.js:

import React from "react";
import ReactDOM from "react-dom/client";

class Test extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hello: "World!" };
  }
  componentDidMount() {
    this.setState({hello:"hello!"})
  }
  render() {
    return (
      <div>
        <h1>
          hello.org, Hello
          {this.state.hello}
        </h1>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(
  document.getElementById("root")
);
root.render(<Test />);
```

### 3. Updation

React is a JS library that helps create Active web pages easily. Now active web pages are specific pages that behave according to their user. For example, let's take the hello {IDE} webpage, the webpage acts differently with each user. User A might write some code in C in the Light Theme while another User may write Python

code in the Dark Theme all at the same time. This dynamic behavior that partially depends upon the user itself makes the webpage an Active webpage. Now how can this be related to Updation? Updation is the phase where the states and props of a component are updated followed by some user events such as clicking, pressing a key on the keyboard, etc. The following are the descriptions of functions that are invoked at different points of the Updation phase.

- `getDerivedStateFromProps`
- `setState()` Function
- `shouldComponentUpdate()`
- `getSnapshotBeforeUpdate()` Method
- `componentDidUpdate()`

#### **getDerivedStateFromProps:**

`getDerivedStateFromProps(props, state)` is a static method that is called just before `render()` method in both mounting and updating phase in React. It takes updated props and the current state as arguments.

```
static getDerivedStateFromProps(props, state)
{
    if(props.name !== state.name){
        //Change in props
        return{
            name: props.name
        };
    }
    return null; // No change to state
}
```

#### **setState()**

This is not particularly a Lifecycle function and can be invoked explicitly at any instant. This function is used to update the state of a component. You may refer to [this article](#) for detailed information.

```
this.setState((prevState, props) => ({
  counter: prevState.count + props.diff
}));
```

### **setState Example:**

```
// Filename - index.js

import React from "react";
import ReactDOM from "react-dom/client";

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0,
    };
  }

  increment = () => {
    this.setState((prevState) => ({
      count: prevState.count + 1,
    }));
  };

  decrement = () => {
    this.setState((prevState) => ({
      count: prevState.count - 1,
    }));
  };

  render() {
```

```

        return (
            <div>
                <h1>
                    The current count is :{" "}
                    {this.state.count}
                </h1>
                <button onClick={this.increment}>
                    Increase
                </button>
                <button onClick={this.decrement}>
                    Decrease
                </button>
            </div>
        );
    }
}

const root = ReactDOM.createRoot(
    document.getElementById("root")
);
root.render(
    <React.StrictMode>
        <App />
    </React.StrictMode>
);

```

### shouldComponentUpdate()

By default, every state or props update re-renders the page but this may not always be the desired outcome, sometimes it is desired that updating the page will not be repainted. The `shouldComponentUpdate()` Function fulfills the requirement by letting React know whether the component's output will be affected by the update or not. `shouldComponentUpdate()` is invoked before rendering an already mounted component when new props or states are being received. If returned false then the subsequent steps of rendering will not be carried out. This function can't be used in the case of



forceUpdate(). The Function takes the new Props and new State as the arguments and returns whether to re-render or not.

`shouldComponentUpdate(nextProps, nextState)`

It returns true or false, if false then

`render()`, `componentWillUpdate()` and

`componentDidUpdate()` method does not gets invoked.

### **getSnapshotBeforeUpdate() Method**

The `getSnapshotBeforeUpdate()` method is invoked just before the DOM is being rendered. It is used to store the previous values of the state after the DOM is updated.

`getSnapshotBeforeUpdate(prevProps, prevState)`

### **componentDidUpdate() Function**

Similarly this function is invoked after the component is rerendered i.e. this function gets invoked once after the `render()` function is executed after the updation of State or Props.

`componentDidUpdate(prevProps, prevState, snapshot)`

## **4. Unmounting**

This is the final phase of the lifecycle of the component which is the phase of unmounting the component from the DOM. The following function is the sole member of this phase.

### **componentWillUnmount() Function:**

This function is invoked before the component is finally unmounted from the DOM i.e. this function gets invoked once before the component is removed from the page and this denotes the end of the lifecycle.

### **componentWillUnmount() Example**

```

import React from "react";
class ComponentOne extends React.Component {
  // Defining the componentWillUnmount method
  componentWillUnmount() {
    alert("The component is going to be
unmounted");
  }

  render() {
    return <h1>Hello hello!</h1>;
  }
}

class App extends React.Component {
  state = { display: true };
  delete = () => {
    this.setState({ display: false });
  };

  render() {
    let comp;
    if (this.state.display) {
      comp = <ComponentOne />;
    }
    return (
      <div>
        {comp}
        <button onClick={ this.delete }>
          Delete the component
        </button>
      </div>
    );
  }
}

export default App;

```

## Implementing the Component Lifecycle methods

Let us now see one final example to finish the article while revising what's discussed above.

First, create a react app and edit your **index.js** file from the src folder.

```
// Filename - src/index.js:

import React from "react";
import ReactDOM from "react-dom/client";

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

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

  changeState() {
    this.setState({ hello: "Geek!" });
  }

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

```

        </h2>
      </div>
    );
  }

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

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

const root = ReactDOM.createRoot(
  document.getElementById("root")
);
root.render(<Test />);

```

**Output:** This output will be visible on the **http://localhost:3000** on the browser window.

# Differences between Functional Components and Class Components

## Functional Components

Functional components are some of the more common components that will come across while working in React. These are simply JavaScript functions. We can

create a functional component to React by writing a JavaScript function.

### **Syntax:**

```
const Car={()=> {  
    return <h2>Hi, I am also a Car!</h2>;  
}}
```

### **Counter using Functional Components**

### **Example:**

```

import React, { useState } from "react";

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

  const increase = () => {
    setCount(count + 1);
  }

  return (
    <div style={{ margin: '50px' }}>
      <h1>Welcome to hello for hello </h1>
      <h3>Counter App using Functional
Component : </h3>
      <h2>{count}</h2>
      <button onClick={increase}>Add</button>
    </div>
  )
}

export default FunctionalComponent;

```

## Class Component

This is the bread and butter of most modern web apps built in ReactJS. These components are simple classes (made up of multiple functions that add functionality to the application).

### Syntax:

```

class Car extends React.Component {
  render() {
    return <h2>Hi, I am a Car!</h2>;
  }
}

```

## Counter using Class Components

## Example:

```
import React, { Component } from "react";

class ClassComponent extends React.Component {
  constructor() {
    super();
    this.state = {
      count: 0
    };
    this.increase = this.increase.bind(this);
  }

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

  render() {
    return (
      <div style={{ margin: '50px' }}>
        <h1>Welcome to hello for hello </
h1>
        <h3>Counter App using Class
Component : </h3>
        <h2> {this.state.count}</h2>
        <button onClick={ this.increase }>
Add</button>

      </div>
    )
  }
}

export default ClassComponent;
```

In the above example, for functional components, we use hooks (useState) to manage the state. If you write a function component and realize you need to add some



state to it, previously you had to convert it to a class component. Now you can use a Hook inside the existing function component to manage the state and no need to convert it into the Class component. Hooks are a new addition to React 16.8. They let you use state and other React features without writing a class. Instead of Classes, one can use Hooks in the Functional component as this is a much easier way of managing the state. Hooks can only be used in functional components, not in-class components.

## **Functional Components vs Class Components:**

<b><u>Functional</u></b> <b><u>Components</u></b>	<b><u>Class</u></b> <b><u>Components</u></b>
A functional component is just a plain JavaScript pure function that accepts props as an argument and returns a React element(JSX).	A class component requires you to extend from React.Component and create a render function that returns a React element.
There is no render method used in functional components.	It must have the render() method returning JSX (which is syntactically similar to HTML)
Functional components run from top to bottom and once the function is returned it can't be kept alive.	The class component is instantiated and different lifecycle method is kept alive and is run and invoked depending on the phase of the class component.

Also known as Stateless components as they simply accept data and display them in some form, they are mainly responsible for rendering UI.

React lifecycle methods (for example, `componentDidMount`) cannot be used in functional components.

Hooks can be easily used in functional components to make them Stateful.

Example:

```
const [name,SetName]=  
  React.useState(' ')
```

Constructors are not used.

Also known as Stateful components because they implement logic and state.

React lifecycle methods can be used inside class components (for example, `componentDidMount`).

It requires different syntax inside a class component to implement hooks.

Example:

```
constructor(props) {  
  super(props);  
  this.state = {name:  
    ' '}  
}
```

Constructor is used as it needs to store state.

## ReactJS Methods as Props

In this article, we will learn about props and passing methods as props. We will also discuss how we can use the child components to pass data to parent components using methods as props.

### What are props?

We know that everything in **ReactJS** is a **component**

and to pass in data to these components, **props** are used. Whenever we call child components from parents we can pass data as props. This helps the parent component communicate with the child.

Although passing in props like this is great, it surely lacks flexibility in an application. For example, we cannot let the child communicate with the parent in this way. This, nonetheless, can be done by passing methods as props in **ReactJS**.

### **Passing methods as props**

We will learn passing props as methods with the help of an example. To use a method as a prop all the steps are described below order wise:

**Step 1:** Create a new react application using the following command.

```
npx create-react-app
```

**Step 2:** We will create components in our file namely App.js. After using this ParentComponent.js and ChildComponent.

**Step 3:** Write the following code in respective files.

- **App.js:** This file imports our ParentComponent and renders it on the page.

- **ParentComponent.js:** This file sends methods as props to child component.
- **ChildComponent:** This file calls the method passed from parent as props.

## Javascript

```
import './App.css';
import React from 'react';

// imports component
import ParentComponent from './components/
ParentComponent';

function App() {
  return (
    <div className="App">
      <h1>-----METHODS AS
PROPS-----</h1>
      <ParentComponent />

    </div>
  );
}

export default App;
```

## Javascript

```

import React, { Component } from 'react';
import ChildComponent from './ChildComponent';

class ParentComponent extends Component {
  constructor(props) {
    super(props);

    this.state = {
      parentName: 'Parent'
    }

    this.greetParent =
this.greetParent.bind(this)
  }

  greetParent() {
    alert(`Hello ${this.state.parentName}`)
  }

  render() {
    return (
      <div>
        <ChildComponent
greetHandler={ this.greetParent } />
      </div>
    )
  }
}

export default ParentComponent;

```

## JavaScript

```
import React from 'react';

function ChildComponent(props) {
  return (
    <div>
      <button onClick={() =>
props.greetHandler()}>
        Greet Parent
      </button>
    </div>
  )
}

export default ChildComponent;
```

**Output:**

-----METHODS AS PROPS-----

Greet Parent

## Passing parameters to parents in methods as props

Till now we learned how to pass methods as props but now we will use the same technique to pass parameters in these methods as props.

**Note:** We will be using the same files made in the previous example. Just some code modification will be there.

**Example:** Write the following code in the respective files.

- **App.js:** This file imports our ParentComponent and renders it on the page.

- **ParentComponent.js:** This file sends methods as props to child component.
- **ChildComponent:** This file calls the method passed from parent as props and passes a parameter to the parent.

## Javascript

```
// App.js

import './App.css';
import React from 'react';

// imports component
import ParentComponent from './components/ParentComponent';

function App() {
  return (
    <div className="App">
      <h1>-----METHODS AS
PROPS-----</h1>
      <ParentComponent />

    </div>
  );
}

export default App;
```

## Javascript

```
// ParentComponent.js

import React, { Component } from 'react';
import ChildComponent from './ChildComponent';

class ParentComponent extends Component {
  constructor(props) {
    super(props);

    this.greetParent =
this.greetParent.bind(this)
  }

  greetParent(name) {
    alert(`Hello ${name}`)
  }

  render() {
    return (
      <div>
        <ChildComponent
greetHandler={ this.greetParent } />
      </div>
    )
  }
}

export default ParentComponent;
```

## Javascript



```
// ChildComponent.js

import React from 'react';

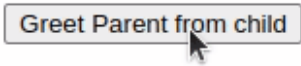
function ChildComponent(props) {
  return (
    <div>
      <button onClick={() =>
props.greetHandler("Child")}>
        Greet Parent from child
      </button>
    </div>
  )
}

export default ChildComponent;
```

**Output:**

---

-----**METHODS AS PROPS**-----



## ReactJS PropTypes

Last Updated : 14 Aug, 2023

Improve

In this article, we will learn about PropTypes in react and how we can use these PropTypes so that we can create strict rules for the data being passed in the props.

## Issues with passing props

We passed different types of information like integers, strings, arrays, etc. as props to the components. We can either create defaultProps or have passed props directly as attributes to the components. We do not have a check on what type of values we are getting inside our Component through props. For larger Apps, it is always a good practice to validate the data we are getting through props. This will help in debugging and also helps in avoiding bugs in the future. Let us see how to do this.

## React propTypes

Before the release of React 15.5.0 version propTypes was available in the react package but in later versions of React have to add a dependency in your project. You can add the dependency to your project by using the command given below:

```
npm i prop-types
```

We can use the propTypes for validating any data we are receiving from props. But before using it we will have to import it. Add the below line at the top of your index.js file :

```
import PropTypes from 'prop-types';
```

Once we have imported propTypes we are ready to work with them. Just like defaultProps, propTypes are also objects where keys are the prop names and values

are their types.

### **Syntax:**

```
ComponentClassName.propTypes{  
    propName1 : PropTypes.string,  
    propName2 : PropTypes.bool,  
    propName3 : PropTypes.array,  
    .  
    .  
    propNameN : PropTypes.anyOtherType  
}
```

In the above Syntax, the *ComponentClassName* is the name of the class of Component, *anyOtherType* can be any type that we are allowed to pass as props. For the props which do not validate the type of data specified by *propTypes*, a warning on the console will occur.

Let us see a complete program that uses *propTypes* for validation for a better understanding:

**Example:** Write the following code in *index.js* file of your react application

```
import PropTypes from 'prop-types';  
import React from 'react';  
import ReactDOM from 'react-dom/client';  
  
// Component  
class ComponentExample extends React.Component{  
    render(){
```

```

    return (
      <div>

        { /* printing all props */ }
        <h1>
          { this.props.arrayProp }
          <br />

          { this.props.stringProp }
          <br />

          { this.props.numberProp }
          <br />

          { this.props.boolProp }
          <br />
        </h1>
      </div>
    );
  }
}

// Validating prop types
ComponentExample.propTypes = {
  arrayProp: PropTypes.array,
  stringProp: PropTypes.string,
  numberProp: PropTypes.number,
  boolProp: PropTypes.bool,
}

// Creating default props
ComponentExample.defaultProps = {

  arrayProp: ['Ram', 'Shyam', 'Raghav'],
  stringProp: "hello",
  numberProp: "10",
  boolProp: true,
}

const root =
ReactDOM.createRoot(document.getElementById("root"))
);
root.render(

```

```
    <React.StrictMode>
      <ComponentExample />
    </React.StrictMode>
  );
```

### **Explanation:**

You can see in the above program that we are passing the prop named `numberProp` as a string but validating it as a number. Still, everything is rendered perfectly on the browser but our browser console has a warning message. This message clearly tells us that the prop named `numberProp` was expected to contain a numeric value but instead, a string value is passed.

**Note:** In recent versions of React the **React.PropTypes** is moved to a different package, and we will have to install that package separately in order to use it. Please go to <https://www.npmjs.com/package/prop-types> link for installation instructions.

## **ReactJS Props – Set 1**

Till now we were working with components using static data only. In this article, we will learn about how we can pass information to a Component.

### **What is props?**

React allows us to pass information to a Component using something called **props** (which stands for properties). Props are objects which can be used inside

a component.

## Passing and Accessing props

We can pass props to any component as we declare attributes for any HTML tag.

### Syntax:

```
// Passing Props
```

```
<DemoComponent sampleProp = "HelloProp" />
```

In the above code snippet, we are passing a **prop** named `sampleProp` to the component named `DemoComponent`. This prop has the value “HelloProp”. Let us now see how can we access these props. We can access any props inside from the component’s class to which the props is passed.

### Syntax:

```
// Accessing props
```

```
this.props.propName;
```

The ‘`this.props`’ is a kind of global object which stores all of a component’s props. The `propName`, that is the names of props are keys of this object.

Let us see an example where we will implement passing and accessing props.

In the below example, we will use a class-based

component to illustrate the props. But we can also pass props to function-based components and going to pass in the below example.

To access a prop from a function we do not need to use the 'this' keyword anymore. Functional components accept props as parameters and can be accessed directly.

Open your react project and edit the **index.js** file in the src folder:

**Example:** Write the following code in your index.js file.

```

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

// sample component to illustrate props
class DemoComponent extends React.Component{
  render(){
    return(
      <div>
        { /*accessing information from
props */}
        <h2>Hello {this.props.user}</
h2>
        <h3>Welcome to hello</h3>
      </div>
    );
  }
}

const root =
ReactDOM.createRoot(document.getElementById("root")
);
root.render(
  <React.StrictMode>
    <DemoComponent />
  </React.StrictMode>
);

```

Open your react project and edit the **index.js** file in the src folder:

**Example:** Write the following code in your index.js file.



```

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

// functional component to illustrate props
function DemoComponent(props) {
  return (
    <div>
      { /*accessing information from props */ }
      <h2>Hello {props.user}</h2>
      <h3>Welcome to hello</h3>
    </div>
  );
}

const root =
ReactDOM.createRoot(document.getElementById("root"))
);
root.render(
  <React.StrictMode>
    <DemoComponent user="hello"/>
  </React.StrictMode>
);

```

## Passing information from one component to another

This is one of the coolest features of React. We can make components to interact among themselves. We will consider two components Parent and Children to explain this. We will pass some information as props from our Parent component to the Child component. **We can pass as many props as we want to a component.**

Look at the below code:

Open your react project and edit the **index.js** file in the src folder:

**Example:** Write the following code in your index.js file.

```

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

// Parent Component
class Parent extends React.Component{
  render(){
    return(
      <div>
        <h2>You are inside Parent
Component</h2>
        <Child name="User" userId =
"5555"/>
      </div>
    );
  }
}

// Child Component
class Child extends React.Component{
  render(){
    return(
      <div>
        <h2>Hello, {this.props.name}</
h2>
        <h3>You are inside Child
Component</h3>
        <h3>Your user id is:
{this.props.userId}</h3>
      </div>
    );
  }
}

const root =
ReactDOM.createRoot(document.getElementById("root")
);
root.render(
  <React.StrictMode>
    <Parent/>
  </React.StrictMode>
);

```

**Output:** See the output of this program

**You are inside Parent Component**

**Hello, User**

**You are inside Child Component**

**Your user id is: 5555**

## **What is this.props in React?**

So we have seen props in React and also have learned about how props are used, how they can be passed to a component, how they are accessed inside a component, and much more. We have used the thing “this.props.propName” very often in this complete scenario to access props. Let us now check what is actually being stored in this. We will console.log “this.props” in the above program inside the child component and will try to observe what is logged into the console. Below is the modified program with console.log() statement:

Open your react project and edit the **index.js** file in the src folder:

**Example:** Write the following code in your index.js file.

```

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

// Parent Component
class Parent extends React.Component{
  render(){
    return(
      <div>
        <h2>You are inside Parent
Component</h2>
        <Child name="User" userId =
"5555"/>
      </div>
    );
  }
}

// Child Component
class Child extends React.Component{
  render(){
    console.log(this.props);
    return(
      <div>
        <h2>Hello, {this.props.name}</
h2>
        <h3>You are inside Child
Component</h3>
        <h3>Your user id is:
{this.props.userId}</h3>
      </div>
    );
  }
}

const root =
ReactDOM.createRoot(document.getElementById("root")
);
root.render(
  <React.StrictMode>
    <Parent/>
  </React.StrictMode>
);

```

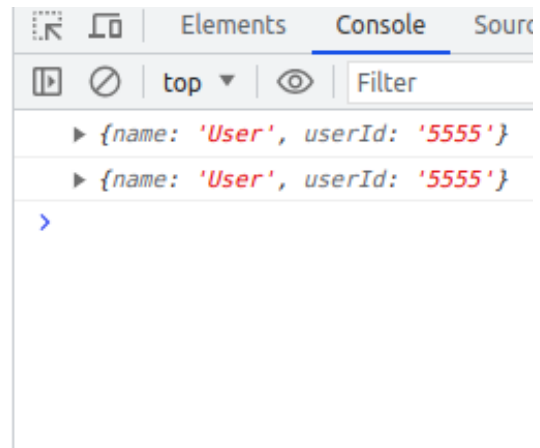
## Output:

**You are inside Parent  
Component**

**Hello, User**

**You are inside Child Component**

**Your user id is: 5555**



You can clearly see in the above image that in the console it is shown that the **this.props** is an object and it contains all of the props passed to the child component. The props name of the child component are keys of this object and their values are values of these keys. So, it is clear now that whatever information is carried to a component using props is stored inside an object.

**Note:** Props are read-only. We are not allowed to modify the content of a prop. Whatever the type of Component is – functional or class-based, none of them is allowed to modify their props.

## ReactJS Props – Set 2

In our previous article [ReactJS Props – Set 1](#) we discussed props, passing and accessing props, passing props from one component to another, etc. In this article, we will continue our discussion on props.

So, what if we want to pass some default information using props to our components? React allows us to do so. React provides us with something called

defaultProps for this purpose. Let's see about this in detail:

**defaultProps:** The defaultProps is a method that we can use to store as much information as we want for a particular class. And this information can be accessed from anywhere inside that particular class. Every piece of information you add inside the defaultProps, will be added as the prop of that class. It might seem confusing at this point. Let's look at a program where we will be using defaultProps to create some props for a class.

Open your react project directory and edit the **App.js** file from src folder:

**src/App.js:**

```

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

// Component
class ExampleClass extends React.Component {
  render() {
    return (
      <div>
        { /* using default prop - title */ }
        <h1>This is { this.props.title}'s
Website!</h1>
      </div>
    );
  }
}

// Creating default props for
// ExampleClass Component
ExampleClass.defaultProps = {
  title: "hello"
}

ReactDOM.render(
  <ExampleClass />,
  document.getElementById("root")
);

```

We can also pass arrays as props, instead of passing single elements. Let's just see how it is done in the below program:

Open your react project directory and edit the **App.js** file from src folder:

**src/App.js:**



```

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

// Component
class ExampleClass extends React.Component {
  render() {
    return (
      <div>
        {/* accessing array prop directly */}
        <h1>The names of students are:
        {this.props.names}</h1>
      </div>
    );
  }
}

// Passing an array as prop
ExampleClass.defaultProps = {
  names: ['Ram', 'Shyam', 'Raghav']
}

ReactDOM.render(
  <ExampleClass />,
  document.getElementById("root")
);

```

Open your react project directory and edit the **App.js** file from src folder:

**src/App.js:**

```

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

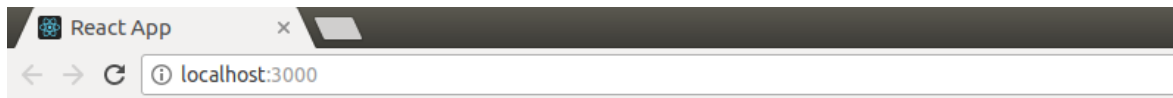
// Component
class ExampleClass extends React.Component {
  render() {
    return (
      <div>
        { /* iterating over array using
map() */ }
        <h1>{this.props.names.map(
          function namesIterator(item, i)
{
            return (
              "Student " + (i + 1) +
": " +
              item +
              ((i !== 2) ? ', ' :
'\n')
            )
          }
        )}</h1>
      </div>
    );
  }
}

// Passing an array as prop
ExampleClass.defaultProps = {
  names: ['Ram', 'Shyam', 'Raghav']
}

ReactDOM.render(
  <ExampleClass />,
  document.getElementById("root")
);

```

**Output:**



**Student 1: Ram, Student 2: Shyam, Student 3: Raghav**

You can see in the above program how we are iterating over the array passed to the component `ExampleClass` using `map()`.

That's all for this article. In our next article, we will see how to validate types in props or how to perform type-checking.

## ReactJS Unidirectional Data Flow

### What is Unidirectional Data Flow?

Unidirectional data flow is a technique that is mainly found in functional reactive programming. It is known as one-way data flow, which means the data has one, and only one way to be transferred to other parts of the application. In essence, this means child components are not able to update the data that is coming from the parent component. In React, data coming from a parent is called **props**

## Data flow in React

React doesn't support bi-directional binding to make sure you are following a clean data flow architecture. The major benefit of this approach is that data flows throughout your app in a single direction, giving you better control over it. In terms of React it means:

- state is passed to the view and to child components
- actions are triggered by the view
- actions can update the state
- the state change is passed to the view and to child components

**Note:** The view is a result of the application state. State changes when actions happen. When actions happen, the state is updated.

## Benefits of One-way data binding/Unidirectional Data Flow

- Easier to debug, as we know what data is coming from where.
- Less prone to errors, as we have more control over our data.
- More efficient, as the library knows what the boundaries are of each part of the system.

## Effects of state change in React

In React, a state is always owned by one component. Any changes made by this state can only affect the

components below it, i.e its children. Changing state on a component will never affect its parent or its siblings, only the children will be affected. This is the main reason that the state is often moved up in the component tree so that it can be shared between the components that need to access it.

## ReactJS State

The State is a way to store and manage the information or data while creating a React Application. The state is a **JavaScript object** that contains the real-time data or information on the webpage.

### Table of Content

- [What is React State?](#)
- [Creating State Object](#)
- [Conventions of Using State in React](#)
- [Updating State in React](#)

### What is a React State?

The state in React is an instance of the **React Component Class** that can be defined as an object of a set of **observable** properties that control the behavior of the **component**.

In other words, the State of a component is an object that holds some information that may change over the lifetime of the component.

# Creating State Object

Creating a state is essential to building dynamic and interactive components.

We can create a state object within the **constructor** of the class component.

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: 'Ford', // Example property in the
state
    };
  }

  render() {
    return (
      <div>
        <h1>My Car</h1>
        { /* Other component content */ }
      </div>
    );
  }
}

export default MyComponent;
```

## Conventions of Using State in React

- The state of a component should prevail throughout its lifetime, **thus we must first have some initial state**, to do so we should define the State in the **constructor** of the component's class.
- **The state should never be updated explicitly.** React uses an observable object as the state that

observes what changes are made to the state and helps the component behave accordingly.

- React provides its own method **setState()**. `setState()` method takes a single parameter and expects an object which should contain the set of values to be updated. Once the update is done the method implicitly calls the **render()** method to repaint the page. Hence, the correct method of updating the value of a state will be similar to the code below.
- **State updates should be independent.** The state object of a component may contain multiple attributes and React allows to use `setState()` function to update only a subset of those attributes as well as using multiple `setState()` methods to update each attribute value independently.
- The only time we are allowed to define the state explicitly is in the constructor to provide the initial state.

## Updating State in React

In React, a State object can be updated using **`setState()` method**.

React may update multiple `setState()` updates in a single go. Thus using the value of the current state may not always generate the desired result.

```
this.setState((prevState, props) => ({
  counter: prevState.count + props.diff
}));
```

Now let us see an example where we will implement

state in React and use the state to create a counter

## Example

This example demonstrates the use of React JS state creating a simple counter application.

```
// Filename - index.js

import React from "react";
import ReactDOM from "react-dom/client";

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0,
    };
  }

  increment = () => {
    this.setState((prevState) => ({
      count: prevState.count + 1,
    }));
  };

  decrement = () => {
    this.setState((prevState) => ({
      count: prevState.count - 1,
    }));
  };

  render() {
    return (
      <div>
        <h1>
          The current count is :{" "}
          {this.state.count}
        </h1>
        <button onClick={this.increment}>
          Increase
        </button>
      </div>
    );
  }
}
```



```

        </button>
        <button onClick={this.decrement}>
          Decrease
        </button>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(
  document.getElementById("root")
);
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);

```

**Output:**

**The current count is : 7**



## ReactJS State vs Props

In React props are a way to pass the data or properties from parent component to child components while the state is the real-time data available to use within that only component.

States and props are two of the most important

concepts of React and everything in React is based upon them. But before jumping into State and Props we need to know about the relation between components and normal functions.

## Table of Content

- Relation between Components and normal functions in JavaScript
- State in React
- Props in React
- Difference between props and state

## Relation between Components and normal functions in JavaScript

We know that react components are the building blocks that can be reused again and again in building the UI. Before jumping into the main difference between the state and props, let's see how a component in react is related to a normal function in javascript.

### Example:

```
// simple component
class FakeComponent extends React.Component {
  render() {
    return <div>Hello World!</div>
  }
}
// simple javascript function
const FakeFunction = () => console.log('Hello
```

```
World!');
```

In the above code, we declared a simple react component by extending the **React.component** native method and then we simply render a div that contains 'Hello World' inside it as text. After the function we have a simple javascript function inside it which contains a simple console.log that does the same thing inside it, printing 'Hello World!'.

## State in React:

A state is a variable that exists inside a component, that cannot be accessed and modified outside the component, and can only be used inside the component. Works very similarly to a variable that is declared inside a function that cannot be accessed outside the scope of the function in normal javascript. State **Can be modified using this.setState. The state can be asynchronous.** Whenever this.setState is used to change the state class is rerender itself.

## Props in React:

React allows us to pass information to a Component using something called props (which stands for properties). Props are objects which can be used inside a component. Sometimes we need to change the content inside a component. Props come to play in these cases, as they are passed into the component and the user..

## Difference between props and state:

PROPS	STATE
The Data is passed from one component to another. It is Immutable (cannot be modified). Props can be used with state and functional components.  Props are read-only.	The Data is passed within the component only. It is Mutable ( can be modified). The state can be used only with the state components/ class component (Before 16.0). The state is both read and write.