ReactJS tutorial

# React Introduction

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

A ReactJS application is made up of multiple components, each component responsible for outputting a small, reusable piece of HTML code. The components are the heart of all React applications. These Components can be nested with other components to allow complex applications to be built of simple building blocks. ReactJS uses virtual DOM based mechanism to fill data in HTML DOM. The virtual DOM works fast as it only changes individual DOM elements instead of reloading complete DOM every time.

To create React app, we write React components that correspond to various elements. We organize these components inside higher level components which define the application structure. For example, we take a form that consists of many elements like input fields, labels, or buttons. We can write each element of the form as React components, and then we combine it into a higher-level component, i.e., the form component itself. The form components would specify the structure of the form along with elements inside of it.

# Why learn ReactJS?

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

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

# React create-react-app

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

The **create-react-app**  allows you to create and run React project very quickly. This tool is wrapping all of the required dependencies like **Webpack**, **Babel** for React project itself and then you need to focus on writing React code only. This tool sets up the development environment, provides an excellent developer experience, and optimizes the app for production.

## Requirements

The Create React App is maintained by Facebook and can works on any platform, for example, macOS, Windows, Linux, etc. To create a React Project using create-react-app, you need to have installed the following things in your system.

1. Node version >= 8.10
2. NPM version >= 5.6

## Installation

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

### Install React

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

**C:\Users\javatpoint> npm install -g create-react-app**

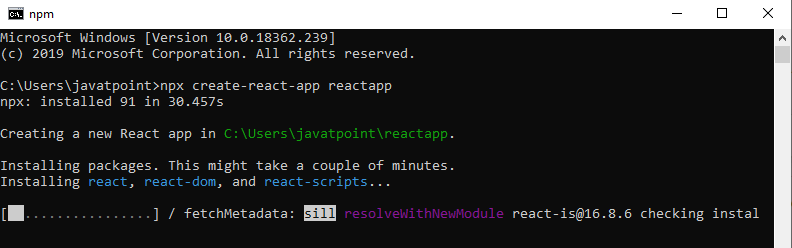
### Create a new React project

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

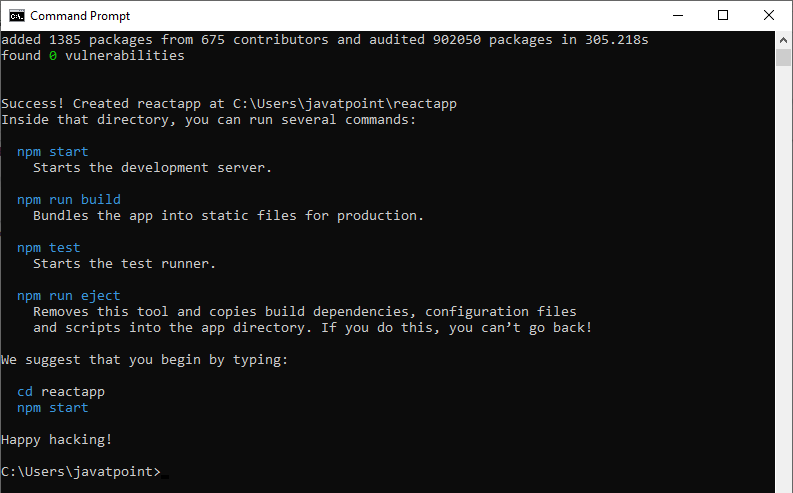
**C:\Users\javatpoint> create-react-app reactproject**

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

**C:\Users\javatpoint> npx create-react-app reactproject**



The above command will take some time to install the React and create a new project with the name "reactproject." Now, we can see the terminal as like below.

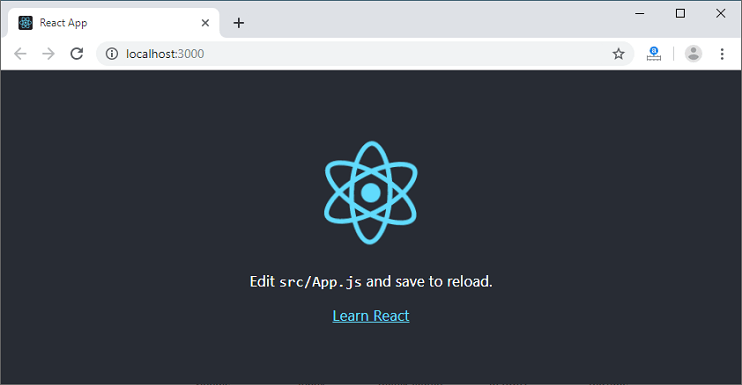


The above screen tells that the React project is created successfully on our system. Now, we need to start the server so that we can access the application on the browser. Type the following command in the terminal window.

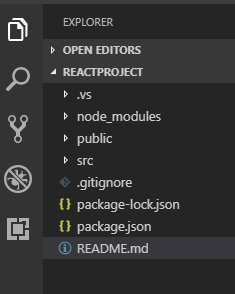
**$ cd Desktop**

**$ npm start**

NPM is a package manager which starts the server and access the application at default server [http://localhost:3000](http://localhost:3000/). Now, we will get the following screen.



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



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

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

### React Environment Setup

Now, open the src >> App.js file and make changes which you want to display on the screen. After making desired changes, save the file. As soon as we save the file, Webpack recompiles the code, and the page will refresh automatically, and changes are reflected on the browser screen. Now, we can create as many components as we want, import the newly created component inside the App.js file and that file will be included in our main index.html file after compiling by Webpack.

Next, if we want to make the project for the production mode, type the following command. This command will generate the production build, which is best optimized.

**$ npm build**

# React Features

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

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

### JSX

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

### Components

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

### One-way Data Binding

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

### Virtual DOM

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

### Simplicity

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

### Performance

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

# Pros and Cons of ReactJS

Today, ReactJS is the highly used open-source JavaScript Library. It helps in creating impressive web apps that require minimal effort and coding. The main objective of ReactJS is to develop User Interfaces (UI) that improves the speed of the apps. There are important pros and cons of ReactJS given as following:

### Advantage of ReactJS

**1. Easy to Learn and USe**

ReactJS is much easier to learn and use. It comes with a good supply of documentation, tutorials, and training resources. Any developer who comes from a JavaScript background can easily understand and start creating web apps using React in a few days. It is the V(view part) in the MVC (Model-View-Controller) model, and referred to as ?one of the JavaScript frameworks.? It is not fully featured but has the advantage of open-source JavaScript User Interface(UI) library, which helps to execute the task in a better manner.

**2. Creating Dynamic Web Applications Becomes Easier**

To create a dynamic web application specifically with HTML strings was tricky because it requires a complex coding, but React JS solved that issue and makes it easier. It provides less coding and gives more functionality. It makes use of the JSX(JavaScript Extension), which is a particular syntax letting HTML quotes and HTML tag syntax to render particular subcomponents. It also supports the building of machine-readable codes.

**3. Reusable Components**

A ReactJS web application is made up of multiple components, and each component has its own logic and controls. These components are responsible for outputting a small, reusable piece of HTML code which can be reused wherever you need them. The reusable code helps to make your apps easier to develop and maintain. These Components can be nested with other components to allow complex applications to be built of simple building blocks. ReactJS uses virtual DOM based mechanism to fill data in HTML DOM. The virtual DOM works fast as it only changes individual DOM elements instead of reloading complete DOM every time.

**4. Performance Enhancement**

ReactJS improves performance due to virtual DOM. The DOM is a cross-platform and programming API which deals with HTML, XML or XHTML. Most of the developers faced the problem when the DOM was updated, which slowed down the performance of the application. ReactJS solved this problem by introducing virtual DOM. The React Virtual DOM exists entirely in memory and is a representation of the web browser's DOM. Due to this, when we write a React component, we did not write directly to the DOM. Instead, we are writing virtual components that react will turn into the DOM, leading to smoother and faster performance.

**5. The Support of Handy Tools**

React JS has also gained popularity due to the presence of a handy set of tools. These tools make the task of the developers understandable and easier. The React Developer Tools have been designed as Chrome and Firefox dev extension and allow you to inspect the React component hierarchies in the virtual DOM. It also allows you to select particular components and examine and edit their current props and state.

**6. Known to be SEO Friendly**

Traditional JavaScript frameworks have an issue in dealing with SEO. The search engines generally having trouble in reading JavaScript-heavy applications. Many web developers have often complained about this problem. ReactJS overcomes this problem that helps developers to be easily navigated on various search engines. It is because React.js applications can run on the server, and the virtual DOM will be rendering and returning to the browser as a regular web page.

**7. The Benefit of Having JavaScript Library**

Today, ReactJS is choosing by most of the web developers. It is because it is offering a very rich JavaScript library. The JavaScript library provides more flexibility to the web developers to choose the way they want.

**8. Scope for Testing the Codes**

ReactJS applications are extremely easy to test. It offers a scope where the developer can test and debug their codes with the help of native tools.

### Disadvantage of ReactJS

**1. The high pace of development**

The high pace of development has an advantage and disadvantage both. In case of disadvantage, since the environment continually changes so fast, some of the developers not feeling comfortable to relearn the new ways of doing things regularly. It may be hard for them to adopt all these changes with all the continuous updates. They need to be always updated with their skills and learn new ways of doing things.

**2. Poor Documentation**

It is another cons which are common for constantly updating technologies. React technologies updating and accelerating so fast that there is no time to make proper documentation. To overcome this, developers write instructions on their own with the evolving of new releases and tools in their current projects.

**3. View Part**

ReactJS Covers only the UI Layers of the app and nothing else. So you still need to choose some other technologies to get a complete tooling set for development in the project.

**4. JSX as a barrier**

ReactJS uses JSX. It's a syntax extension that allows HTML with JavaScript mixed together. This approach has its own benefits, but some members of the development community consider JSX as a barrier, especially for new developers. Developers complain about its complexity in the learning curve.

# React JSX

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

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

### Example

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

**JSX File**

**<div>Hello JavaTpoint</div>**

**Corresponding Output**

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

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

### Why use JSX?

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

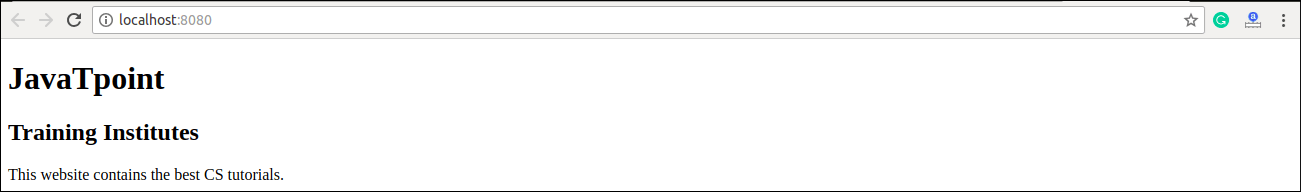
### Nested Elements in JSX

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

**App.JSX**

1. ***import****React, { Component } from 'react';*
2. ***class****App****extends****Component{*
3. *render(){*
4. ***return****(*
5. *<div>*
6. *<h1>JavaTpoint</h1>*
7. *<h2>Training Institutes</h2>*
8. *<p>This website contains the best CS tutorials.</p>*
9. *</div>*
10. *);*
11. *}*
12. *}*
13. *export****default****App;*

**Output:**



### JSX Attributes

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

**Example:**

1. ***import React, { Component } from 'react';***
2. ***class App extends Component{***
3. ***render(){***
4. ***return(***
5. ***<div>***
6. ***<h1>JavaTpoint</h1>***
7. ***<h2>Training Institutes</h2>***
8. ***<p data-demoAttribute = "demo">This website contains the best CS tutorials.</p>***
9. ***</div>***
10. ***);***
11. ***}***
12. ***}***
13. ***export default App;***

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

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

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

**Example**

1. ***import****React, { Component } from 'react';*
2. ***class****App****extends****Component{*
3. *render(){*
4. ***return****(*
5. *<div>*
6. *<h1 className = "hello" >JavaTpoint</h1>*
7. *<p data-demoAttribute = "demo">This website contains the best CS tutorials.</p>*
8. *</div>*
9. *);*
10. *}*
11. *}*
12. *export****default****App;*

**Output:**

JavaTpoint

This website contains the best CS tutorials.

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

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

**Example**

1. ***import****React, { Component } from 'react';*
2. ***class****App****extends****Component{*
3. *render(){*
4. ***return****(*
5. *<div>*
6. *<h1 className = "hello" >{25+20}</h1>*
7. *</div>*
8. *);*
9. *}*
10. *}*
11. *export****default****App;*

**Output:**

**45**

### JSX Comments

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

Example:

1. ***import****React, { Component } from 'react';*
2. ***class****App****extends****Component{*
3. *render(){*
4. ***return****(*
5. *<div>*
6. *<h1 className = "hello" >Hello JavaTpoint</h1>*
7. *{/\* This is a comment in JSX \*/}*
8. *</div>*
9. *);*
10. *}*
11. *}*
12. *export****default****App;*

### JSX Styling

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

Example

1. *import React, { Component } from 'react';*
2. *class App extends Component{*
3. *render(){*
4. *var myStyle = {*
5. *fontSize: 80,*
6. *fontFamily: 'Courier',*
7. *color: '#003300'*
8. *}*
9. *return (*
10. *<div>*
11. *<h1 style = {myStyle}>www.javatpoint.com</h1>*
12. *</div>*
13. *);*
14. *}*
15. *}*
16. *export default App;*

**Output:**



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

Example:

1. ***import****React, { Component } from 'react';*
2. ***class****App****extends****Component{*
3. *render(){*
4. *var i = 5;*
5. ***return****(*
6. *<div>*
7. *<h1>{i == 1 ? 'True!' : 'False!'}</h1>*
8. *</div>*
9. *);*
10. *}*
11. *}*
12. *export****default****App;*

**Output:**

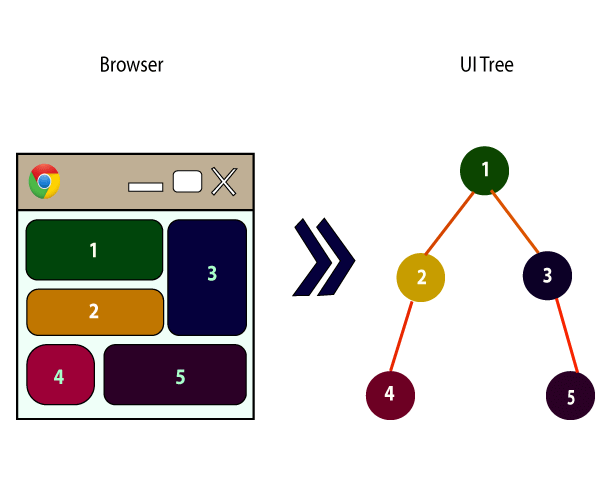
False!

# React Components

Earlier, the developers write more than thousands of lines of code for developing a single page application. These applications follow the traditional DOM structure, and making changes in them was a very challenging task. If any mistake found, it manually searches the entire application and update accordingly. The component-based approach was introduced to overcome an issue. In this approach, the entire application is divided into a small logical group of code, which is known as components.

A Component is considered as the core building blocks of a React application. It makes the task of building UIs much easier. Each component exists in the same space, but they work independently from one another and merge all in a parent component, which will be the final UI of your application.

Every React component have their own structure, methods as well as APIs. They can be reusable as per your need. For better understanding, consider the entire UI as a tree. Here, the root is the starting component, and each of the other pieces becomes branches, which are further divided into sub-branches.



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

1. Functional Components
2. Class Components

## Functional Components

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

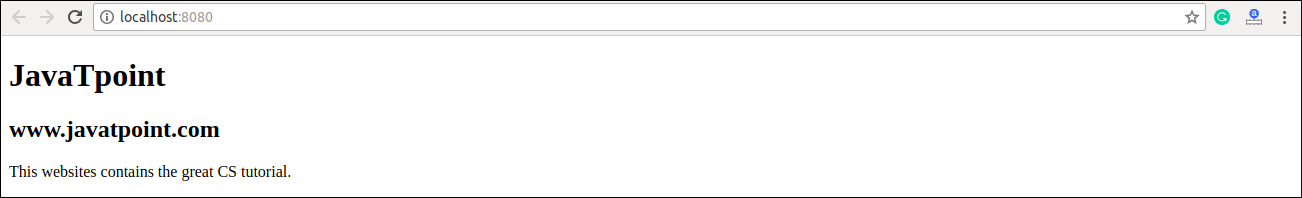
1. *function WelcomeMessage(props) {*
2. ***return****<h1>Welcome to the , {props.name}</h1>;*
3. *}*

The functional component is also known as a stateless component because they do not hold or manage state. It can be explained in the below example.

### Example

1. ***import****React, { Component } from 'react';*
2. ***class****App****extends****React.Component {*
3. *render() {*
4. ***return****(*
5. *<div>*
6. *<First/>*
7. *<Second/>*
8. *</div>*
9. *);*
10. *}*
11. *}*
12. ***class****First****extends****React.Component {*
13. *render() {*
14. ***return****(*
15. *<div>*
16. *<h1>JavaTpoint</h1>*
17. *</div>*
18. *);*
19. *}*
20. *}*
21. ***class****Second****extends****React.Component {*
22. *render() {*
23. ***return****(*
24. *<div>*
25. *<h2>www.javatpoint.com</h2>*
26. *<p>This websites contains the great CS tutorial.</p>*
27. *</div>*
28. *);*
29. *}*
30. *}*
31. *export****default****App;*

**Output:**



## Class Components

Class components are more complex than functional components. It requires you to extend from React. Component and create a render function which returns a React element. You can pass data from one class to other class components. You can create a class by defining a class that extends Component and has a render function. Valid class component is shown in the below example.

1. ***class****MyComponent****extends****React.Component {*
2. *render() {*
3. ***return****(*
4. *<div>This is main component.</div>*
5. *);*
6. *}*
7. *}*

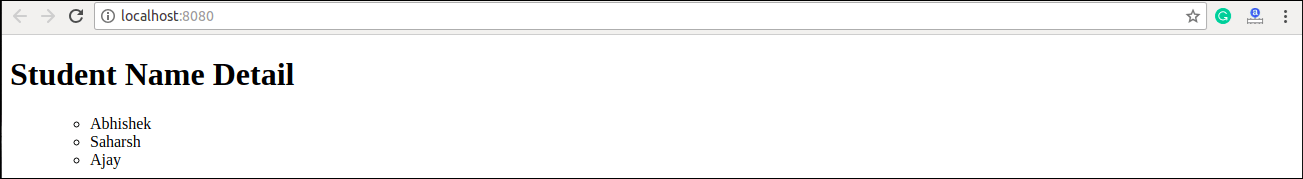
The class component is also known as a stateful component because they can hold or manage local state. It can be explained in the below example.

### Example

In this example, we are creating the list of unordered elements, where we will dynamically insert StudentName for every object from the data array. Here, we are using ES6 arrow syntax (=>) which looks much cleaner than the old JavaScript syntax. It helps us to create our elements with fewer lines of code. It is especially useful when we need to create a list with a lot of items.

1. ***import****React, { Component } from 'react';*
2. ***class****App****extends****React.Component {*
3. *constructor() {*
4. ***super****();*
5. ***this****.state = {*
6. *data:*
7. *[*
8. *{*
9. *"name":"Abhishek"*
10. *},*
11. *{*
12. *"name":"Saharsh"*
13. *},*
14. *{*
15. *"name":"Ajay"*
16. *}*
17. *]*
18. *}*
19. *}*
20. *render() {*
21. ***return****(*
22. *<div>*
23. *<StudentName/>*
24. *<ul>*
25. *{****this****.state.data.map((item) => <List data = {item} />)}*
26. *</ul>*
27. *</div>*
28. *);*
29. *}*
30. *}*
31. ***class****StudentName****extends****React.Component {*
32. *render() {*
33. ***return****(*
34. *<div>*
35. *<h1>Student Name Detail</h1>*
36. *</div>*
37. *);*
38. *}*
39. *}*
40. ***class****List****extends****React.Component {*
41. *render() {*
42. ***return****(*
43. *<ul>*
44. *<li>{****this****.props.data.name}</li>*
45. *</ul>*
46. *);*
47. *}*
48. *}*
49. *export****default****App;*

**Output:**



# React State

The state is an updatable structure that is used to contain data or information about the component. The state in a component can change over time. The change in state over time can happen as a response to user action or system event. A component with the state is known as stateful components. It is the heart of the react component which determines the behavior of the component and how it will render. They are also responsible for making a component dynamic and interactive.

A state must be kept as simple as possible. It can be set by using the **setState()** method and calling setState() method triggers UI updates. A state represents the component's local state or information. It can only be accessed or modified inside the component or by the component directly. To set an initial state before any interaction occurs, we need to use the **getInitialState()** method.

**For example**, if we have five components that need data or information from the state, then we need to create one container component that will keep the state for all of them.

### Defining State

To define a state, you have to first declare a default set of values for defining the component's initial state. To do this, add a class constructor which assigns an initial state using this.state. The '**this.state**' property can be rendered inside **render()** method.

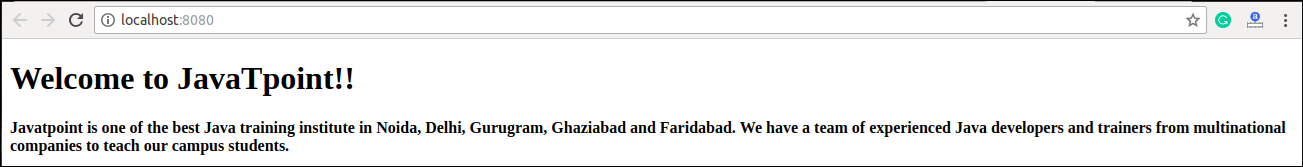
### Example

The below sample code shows how we can create a stateful component using ES6 syntax.

1. ***import****React, { Component } from 'react';*
2. ***class****App****extends****React.Component {*
3. *constructor() {*
4. ***super****();*
5. ***this****.state = { displayBio:****true****};*
6. *}*
7. *render() {*
8. ***const****bio =****this****.state.displayBio ? (*
9. *<div>*
10. *<p><h3>Javatpoint is one of the best Java training institute in Noida, Delhi, Gurugram, Ghaziabad and Faridabad. We have a team of experienced Java developers and trainers from multinational companies to teach our campus students.</h3></p>*
11. *</div>*
12. *) :****null****;*
13. ***return****(*
14. *<div>*
15. *<h1> Welcome to JavaTpoint!! </h1>*
16. *{ bio }*
17. *</div>*
18. *);*
19. *}*
20. *}*
21. *export****default****App;*

To set the state, it is required to call the super() method in the constructor. It is because this.state is uninitialized before the super() method has been called.

**Output**



### Changing the State

We can change the component state by using the setState() method and passing a new state object as the argument. Now, create a new method toggleDisplayBio() in the above example and bind this keyword to the toggleDisplayBio() method otherwise we can't access this inside toggleDisplayBio() method.

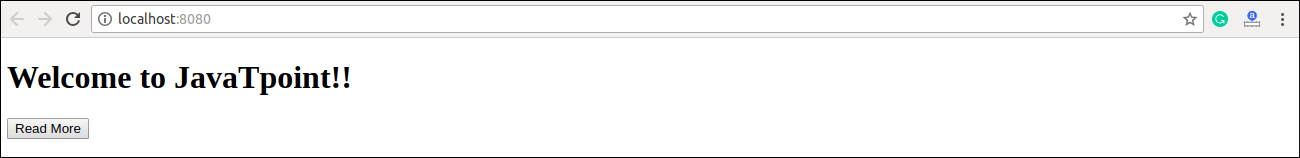
1. **this**.toggleDisplayBio = **this**.toggleDisplayBio.bind(**this**);

### Example

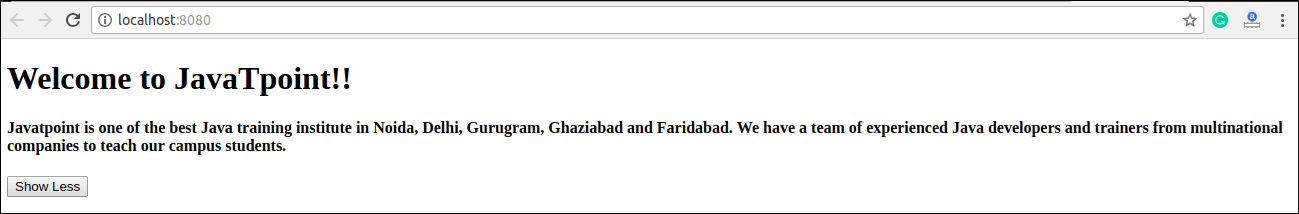
In this example, we are going to add a **button** to the **render**() method. Clicking on this button triggers the toggleDisplayBio() method which displays the desired output.

1. ***import****React, { Component } from 'react';*
2. ***class****App****extends****React.Component {*
3. *constructor() {*
4. ***super****();*
5. ***this****.state = { displayBio:****false****};*
6. *console.log('Component this',****this****);*
7. ***this****.toggleDisplayBio =****this****.toggleDisplayBio.bind(****this****);*
8. *}*
9. *toggleDisplayBio(){*
10. ***this****.setState({displayBio: !****this****.state.displayBio});*
11. *}*
12. *render() {*
13. ***return****(*
14. *<div>*
15. *<h1>Welcome to JavaTpoint!!</h1>*
16. *{*
17. ***this****.state.displayBio ? (*
18. *<div>*
19. *<p><h4>Javatpoint is one of the best Java training institute in Noida, Delhi, Gurugram, Ghaziabad and Faridabad. We have a team of experienced Java developers and trainers from multinational companies to teach our campus students.</h4></p>*
20. *<button onClick={****this****.toggleDisplayBio}> Show Less </button>*
21. *</div>*
22. *) : (*
23. *<div>*
24. *<button onClick={****this****.toggleDisplayBio}> Read More </button>*
25. *</div>*
26. *)*
27. *}*
28. *</div>*
29. *)*
30. *}*
31. *}*
32. *export****default****App;*

**Output:**



When you click the **Read More** button, you will get the below output, and when you click the **Show Less** button, you will get the output as shown in the above image.



# React Props

Props stand for "**Properties**." They are **read-only** components. It is an object which stores the value of attributes of a tag and work similar to the HTML attributes. It gives a way to pass data from one component to other components. It is similar to function arguments. Props are passed to the component in the same way as arguments passed in a function.

Props are **immutable** so we cannot modify the props from inside the component. Inside the components, we can add attributes called props. These attributes are available in the component as **this.props** and can be used to render dynamic data in our render method.

When you need immutable data in the component, you have to add props to **reactDom.render()** method in the **main.js** file of your ReactJS project and used it inside the component in which you need. It can be explained in the below example.

### Example

**App.js**

1. ***import****React, { Component } from 'react';*
2. ***class****App****extends****React.Component {*
3. *render() {*
4. ***return****(*
5. *<div>*
6. *<h1> Welcome to {****this****.props.name } </h1>*
7. *<p> <h4> Javatpoint is one of the best Java training institute in Noida, Delhi, Gurugram, Ghaziabad and Faridabad. </h4> </p>*
8. *</div>*
9. *);*
10. *}*
11. *}*
12. *export****default****App;*

**Main.js**

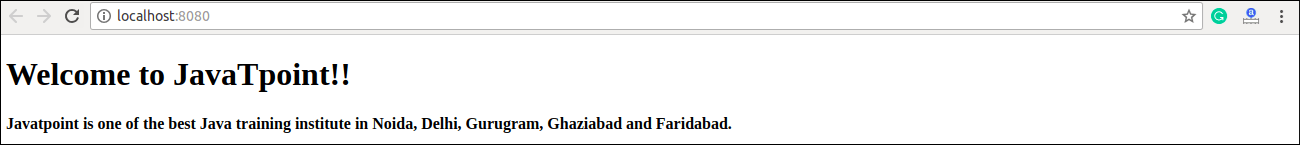
*import React from 'react';*

*import ReactDOM from 'react-dom';*

*import App from './App.js';*

*ReactDOM.render(<App name = "JavaTpoint!!" />, document.getElementById('app'));*

**Output**



### Default Props

It is not necessary to always add props in the reactDom.render() element. You can also set **default** props directly on the component constructor. It can be explained in the below example.

### Example

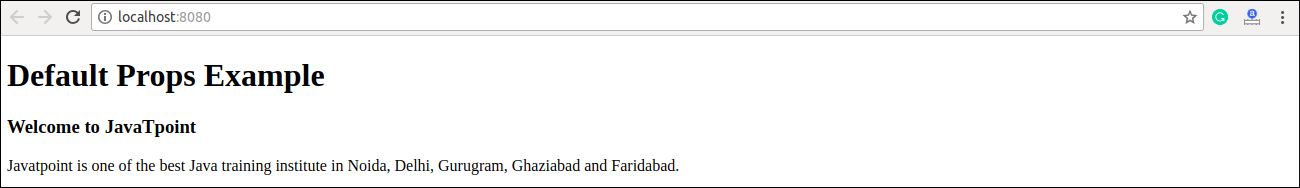
**App.js**

1. ***import****React, { Component } from 'react';*
2. ***class****App****extends****React.Component {*
3. *render() {*
4. ***return****(*
5. *<div>*
6. *<h1>Default Props Example</h1>*
7. *<h3>Welcome to {****this****.props.name}</h3>*
8. *<p>Javatpoint is one of the best Java training institute in Noida, Delhi, Gurugram, Ghaziabad and Faridabad.</p>*
9. *</div>*
10. *);*
11. *}*
12. *}*
13. *App.defaultProps = {*
14. *name: "JavaTpoint"*
15. *}*
16. *export****default****App;*

**Main.js**

1. ***import****React from 'react';*
2. ***import****ReactDOM from 'react-dom';*
3. ***import****App from './App.js';*
5. *ReactDOM.render(<App/>, document.getElementById('app'));*

**Output**



### State and Props

It is possible to combine both state and props in your app. You can set the state in the parent component and pass it in the child component using props. It can be shown in the below example.

Example

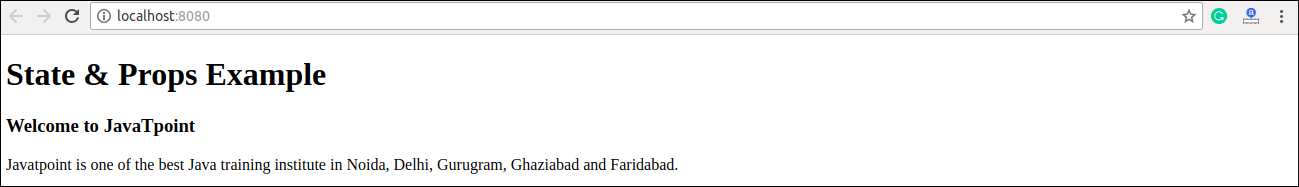
**App.js**

1. ***import****React, { Component } from 'react';*
2. ***class****App****extends****React.Component {*
3. *constructor(props) {*
4. ***super****(props);*
5. ***this****.state = {*
6. *name: "JavaTpoint",*
7. *}*
8. *}*
9. *render() {*
10. ***return****(*
11. *<div>*
12. *<JTP jtpProp = {****this****.state.name}/>*
13. *</div>*
14. *);*
15. *}*
16. *}*
17. ***class****JTP****extends****React.Component {*
18. *render() {*
19. ***return****(*
20. *<div>*
21. *<h1>State & Props Example</h1>*
22. *<h3>Welcome to {****this****.props.jtpProp}</h3>*
23. *<p>Javatpoint is one of the best Java training institute in Noida, Delhi, Gurugram, Ghaziabad and Faridabad.</p>*
24. *</div>*
25. *);*
26. *}*
27. *}*
28. *export****default****App;*

**Main.js**

1. ***import****React from 'react';*
2. ***import****ReactDOM from 'react-dom';*
3. ***import****App from './App.js';*
5. *ReactDOM.render(<App/>, document.getElementById('app'));*

**Output:**



# React Props Validation

Props are an important mechanism for passing the **read-only** attributes to React components. The props are usually required to use correctly in the component. If it is not used correctly, the components may not behave as expected. Hence, it is required to use **props validation** in improving react components.

Props validation is a tool that will help the developers to avoid future bugs and problems. It is a useful way to force the correct usage of your components. It makes your code more readable. React components used special property **PropTypes** that help you to catch bugs by validating data types of values passed through props, although it is not necessary to define components with propTypes. However, if you use propTypes with your components, it helps you to avoid unexpected bugs.

### Validating Props

**App.propTypes** is used for props validation in react component. When some of the props are passed with an invalid type, you will get the warnings on JavaScript console. After specifying the validation patterns, you will set the App.defaultProps.

**Syntax:**

1. **class** App **extends** React.Component {
2. render() {}
3. }
4. Component.propTypes = { /\*Definition \*/};

### ReactJS Props Validator

ReactJS props validator contains the following list of validators.

|  |  |  |
| --- | --- | --- |
| **SN** | **PropsType** | **Description** |
| **1.** | PropTypes.any | The props can be of any data type. |
| **2.** | PropTypes.array | The props should be an array. |
| **3.** | PropTypes.bool | The props should be a boolean. |
| **4.** | PropTypes.func | The props should be a function. |
| **5.** | PropTypes.number | The props should be a number. |
| **6.** | PropTypes.object | The props should be an object. |
| **7.** | PropTypes.string | The props should be a string. |
| **8.** | PropTypes.symbol | The props should be a symbol. |
| **9.** | PropTypes.instanceOf | The props should be an instance of a particular JavaScript class. |
| **10.** | PropTypes.isRequired | The props must be provided. |
| **11.** | PropTypes.element | The props must be an element. |
| **12.** | PropTypes.node | The props can render anything: numbers, strings, elements or an array (or fragment) containing these types. |
| **13.** | PropTypes.oneOf() | The props should be one of several types of specific values. |
| **14.** | PropTypes.oneOfType([PropTypes.string,PropTypes.number]) | The props should be an object that could be one of many types. |

**Example:**

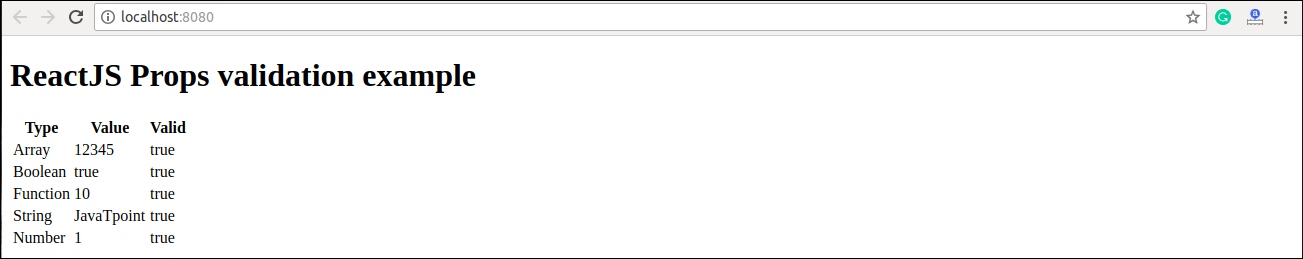
Here, we are creating an App component which contains all the props that we need. In this example, **App.propTypes** is used for props validation. For props validation, you must have to add this line: **import PropTypes from 'prop-types'** in **App.js file**.

**App.js**

1. **import** React, { Component } from 'react';
2. **import** PropTypes from 'prop-types';
3. **class** App **extends** React.Component {
4. render() {
5. **return** (
6. <div>
7. <h1>ReactJS Props validation example</h1>
8. <table>
9. <tr>
10. <th>Type</th>
11. <th>Value</th>
12. <th>Valid</th>
13. </tr>
14. <tr>
15. <td>Array</td>
16. <td>{**this**.props.propArray}</td>
17. <td>{**this**.props.propArray ? "true" : "False"}</td>
18. </tr>
19. <tr>
20. <td>Boolean</td>
21. <td>{**this**.props.propBool ? "true" : "False"}</td>
22. <td>{**this**.props.propBool ? "true" : "False"}</td>
23. </tr>
24. <tr>
25. <td>Function</td>
26. <td>{**this**.props.propFunc(5)}</td>
27. <td>{**this**.props.propFunc(5) ? "true" : "False"}</td>
28. </tr>
29. <tr>
30. <td>String</td>
31. <td>{**this**.props.propString}</td>
32. <td>{**this**.props.propString ? "true" : "False"}</td>
33. </tr>
34. <tr>
35. <td>Number</td>
36. <td>{**this**.props.propNumber}</td>
37. <td>{**this**.props.propNumber ? "true" : "False"}</td>
38. </tr>
39. </table>
40. </div>
41. );
42. }
43. }
44. App.propTypes = {
45. propArray: PropTypes.array.isRequired,
46. propBool: PropTypes.bool.isRequired,
47. propFunc: PropTypes.func,
48. propNumber: PropTypes.number,
49. propString: PropTypes.string,
50. }
51. App.defaultProps = {
52. propArray: [1,2,3,4,5],
53. propBool: **true**,
54. propFunc: function(x){**return** x+5},
55. propNumber: 1,
56. propString: "JavaTpoint",
57. }
58. export **default** App;

**Main.js**

1. **import** React from 'react';
2. **import** ReactDOM from 'react-dom';
3. **import** App from './App.js';
5. ReactDOM.render(<App/>, document.getElementById('app'));



### ReactJS Custom Validators

ReactJS allows creating a custom validation function to perform custom validation. The following argument is used to create a custom validation function.

* **props:** It should be the first argument in the component.
* **propName:** It is the propName that is going to validate.
* **componentName:** It is the componentName that are going to validated again.

**Example**

1. var Component = React.createClass({
2. App.propTypes = {
3. customProp: function(props, propName, componentName) {
4. **if** (!item.isValid(props[propName])) {
5. **return** **new** Error('Validation failed!');
6. }
7. }
8. }
9. })

# State Vs. Props

### State

The state is an updatable structure that is used to contain data or information about the component and can change over time. The change in state can happen as a response to user action or system event. It is the heart of the react component which determines the behavior of the component and how it will render. A state must be kept as simple as possible. It represents the component's local state or information. It can only be accessed or modified inside the component or by the component directly.

### Props

Props are read-only components. It is an object which stores the value of attributes of a tag and work similar to the HTML attributes. It allows passing data from one component to other components. It is similar to function arguments and can be passed to the component the same way as arguments passed in a function. Props are immutable so we cannot modify the props from inside the component.

**Difference between State and Props**

|  |  |  |
| --- | --- | --- |
| **SN** | **Props** | **State** |
| **1.** | Props are read-only. | State changes can be asynchronous. |
| **2.** | Props are immutable. | State is mutable. |
| **3.** | Props allow you to pass data from one component to other components as an argument. | State holds information about the components. |
| **4.** | Props can be accessed by the child component. | State cannot be accessed by child components. |
| **5.** | Props are used to communicate between components. | States can be used for rendering dynamic changes with the component. |
| **6.** | Stateless component can have Props. | Stateless components cannot have State. |
| **7.** | Props make components reusable. | State cannot make components reusable. |
| **8.** | Props are external and controlled by whatever renders the component. | The State is internal and controlled by the React Component itself. |

# What is Constructor?

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

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

**Syntax:**

1. *Constructor(props){*
2. ***super****(props);*
3. *}*

In React, constructors are mainly used for two purposes:

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

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

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

**Example**

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

**App.js**

1. ***import****React, { Component } from 'react';*
3. ***class****App****extends****Component {*
4. *constructor(props){*
5. ***super****(props);*
6. ***this****.state = {*
7. *data: 'www.javatpoint.com'*
8. *}*
9. ***this****.handleEvent =****this****.handleEvent.bind(****this****);*
10. *}*
11. *handleEvent(){*
12. *console.log(****this****.props);*
13. *}*
14. *render() {*
15. ***return****(*
16. *<div className="App">*
17. *<h2>React Constructor Example</h2>*
18. *<input type ="text" value={****this****.state.data} />*
19. *<button onClick={****this****.handleEvent}>Please Click</button>*
20. *</div>*
21. *);*
22. *}*
23. *}*
24. *export****default****App;*

**Main.js**

1. ***import****React from 'react';*
2. ***import****ReactDOM from 'react-dom';*
3. ***import****App from './App.js';*
5. *ReactDOM.render(<App />, document.getElementById('app'));*

**Output:**

When you execute the above code, you get the following output.



The most common question related to the constructor are:

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

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

1. ***class****App****extends****Component {*
2. *render () {*
3. ***return****(*
4. *<p> Name: {****this****.props.name }</p>*
5. *);*
6. *}*
7. *}*

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

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

1. ***class****App****extends****Component {*
2. *constructor(props){*
3. ***this****.fName = "Jhon"; // 'this' is not allowed before super()*
4. *}*
5. *render () {*
6. ***return****(*
7. *<p> Name: {****this****.props.name }</p>*
8. *);*
9. *}*
10. *}*

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

### Arrow Functions

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

1. ***import****React, { Component } from 'react';*
3. ***class****App****extends****Component {*
4. *constructor(props){*
5. ***super****(props);*
6. ***this****.state = {*
7. *data: 'www.javatpoint.com'*
8. *}*
9. *}*
10. *handleEvent = () => {*
11. *console.log(****this****.props);*
12. *}*
13. *render() {*
14. ***return****(*
15. *<div className="App">*
16. *<h2>React Constructor Example</h2>*
17. *<input type ="text" value={****this****.state.data} />*
18. *<button onClick={****this****.handleEvent}>Please Click</button>*
19. *</div>*
20. *);*
21. *}*
22. *}*
23. export **default** App;

We can use a constructor in the following ways:

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

1. ***class****App****extends****Component {*
2. *constructor(props){*
3. *// here, it is setting initial value for 'inputTextValue'*
4. ***this****.state = {*
5. *inputTextValue: 'initial value',*
6. *};*
7. *}*
8. *}*

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

1. ***class****App****extends****Component {*
2. *constructor(props) {*
3. *// when you use 'this' in constructor, super() needs to be called first*
4. ***super****();*
5. *// it means, when you want to use 'this.props' in constructor, call it as below*
6. ***super****(props);*
7. *}*
8. *}*

**3) Initializing third-party libraries**

1. ***class****App****extends****Component {*
2. *constructor(props) {*
4. ***this****.myBook =****new****MyBookLibrary();*
6. *//Here, you can access props without using 'this'*
7. ***this****.Book2 =****new****MyBookLibrary(props.environment);*
8. *}*
9. *}*

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

1. ***class****App****extends****Component {*
2. *constructor(props) {*
4. *// when you need to 'bind' context to a function*
5. ***this****.handleFunction =****this****.handleFunction.bind(****this****);*
6. *}*
7. *}*

# React Component API

ReactJS component is a top-level API. It makes the code completely individual and reusable in the application. It includes various methods for:

* Creating elements
* Transforming elements
* Fragments

Here, we are going to explain the three most important methods available in the React component API.

1. setState()
2. forceUpdate()
3. findDOMNode()

### setState()

This method is used to update the state of the component. This method does not always replace the state immediately. Instead, it only adds changes to the original state. It is a primary method that is used to update the user interface(UI) in response to event handlers and server responses.

#### Note: In the ES6 classes, this.method.bind(this) is used to manually bind the setState() method.

**Syntax:**

1. ***this****.stateState(object newState[, function callback]);*

In the above syntax, there is an optional **callback** function which is executed once setState() is completed and the component is re-rendered.

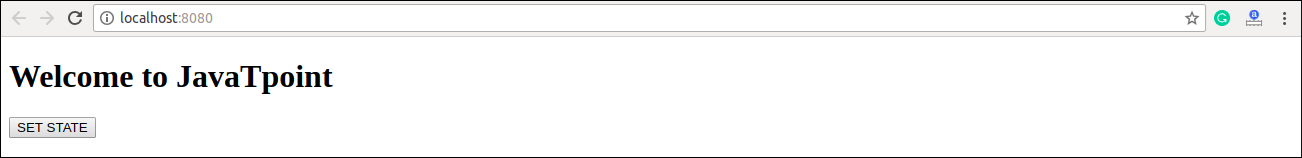
**Example**

1. ***import****React, { Component } from 'react';*
2. ***import****PropTypes from 'prop-types';*
3. ***class****App****extends****React.Component {*
4. *constructor() {*
5. ***super****();*
6. ***this****.state = {*
7. *msg: "Welcome to JavaTpoint"*
8. *};*
9. ***this****.updateSetState =****this****.updateSetState.bind(****this****);*
10. *}*
11. *updateSetState() {*
12. ***this****.setState({*
13. *msg:"Its a best ReactJS tutorial"*
14. *});*
15. *}*
16. *render() {*
17. ***return****(*
18. *<div>*
19. *<h1>{****this****.state.msg}</h1>*
20. *<button onClick = {****this****.updateSetState}>SET STATE</button>*
21. *</div>*
22. *);*
23. *}*
24. *}*
25. *export****default****App;*

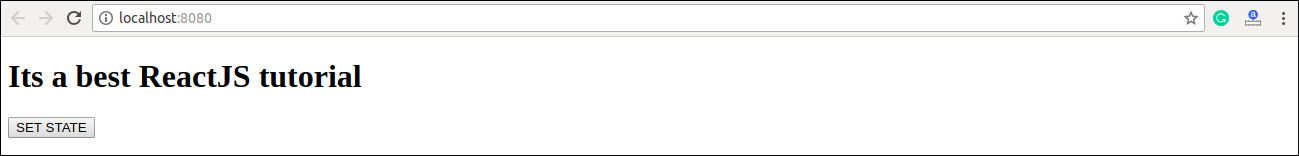
**Main.js**

1. ***import****React from 'react';*
2. ***import****ReactDOM from 'react-dom';*
3. ***import****App from './App.js';*
5. *ReactDOM.render(<App/>, document.getElementById('app'));*

**Output:**



When you click on the **SET STATE** button, you will see the following screen with the updated message.



### forceUpdate()

This method allows us to update the component manually.

**Syntax:**

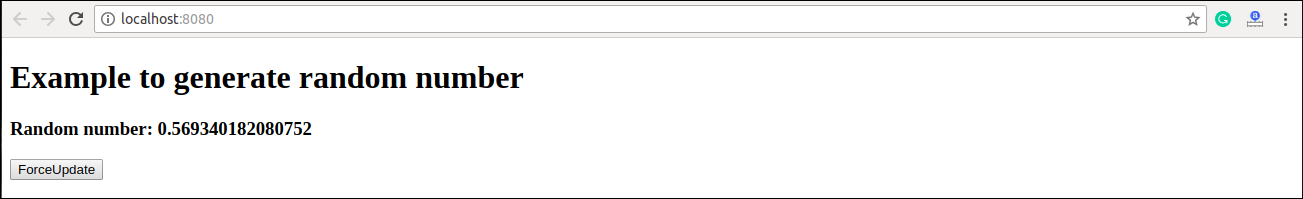
1. Component.forceUpdate(callback);

**Example**

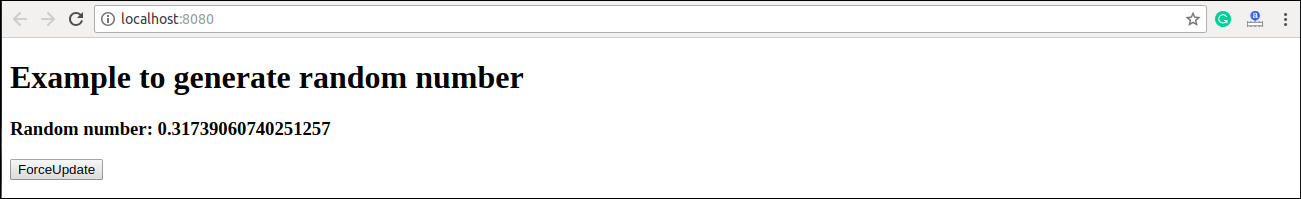
**App.js**

1. ***import****React, { Component } from 'react';*
2. ***class****App****extends****React.Component {*
3. *constructor() {*
4. ***super****();*
5. ***this****.forceUpdateState =****this****.forceUpdateState.bind(****this****);*
6. *}*
7. *forceUpdateState() {*
8. ***this****.forceUpdate();*
9. *};*
10. *render() {*
11. ***return****(*
12. *<div>*
13. *<h1>Example to generate random number</h1>*
14. *<h3>Random number: {Math.random()}</h3>*
15. *<button onClick = {****this****.forceUpdateState}>ForceUpdate</button>*
16. *</div>*
17. *);*
18. *}*
19. *}*
20. *export****default****App;*

**Output:**



Each time when you click on **ForceUpdate** button, it will generate the **random** number. It can be shown in the below image.



### findDOMNode()

For DOM manipulation, you need to use **ReactDOM.findDOMNode()** method. This method allows us to find or access the underlying DOM node.

**Syntax:**

1. *ReactDOM.findDOMNode(component);*

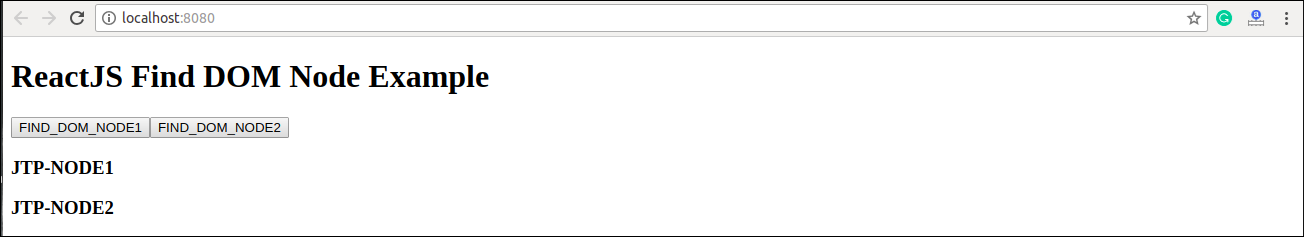
**Example**

For DOM manipulation, first, you need to import this line: **import ReactDOM** from '**react-dom**' in your **App.js** file.

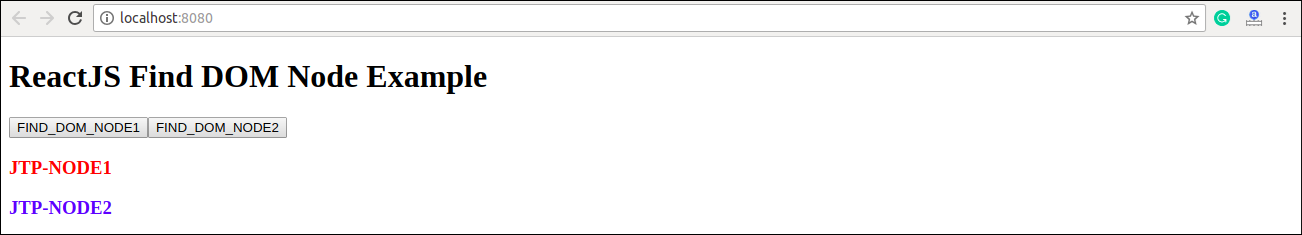
**App.js**

1. ***import****React, { Component } from 'react';*
2. ***import****ReactDOM from 'react-dom';*
3. ***class****App****extends****React.Component {*
4. *constructor() {*
5. ***super****();*
6. ***this****.findDomNodeHandler1 =****this****.findDomNodeHandler1.bind(****this****);*
7. ***this****.findDomNodeHandler2 =****this****.findDomNodeHandler2.bind(****this****);*
8. *};*
9. *findDomNodeHandler1() {*
10. *var myDiv = document.getElementById('myDivOne');*
11. *ReactDOM.findDOMNode(myDivOne).style.color = 'red';*
12. *}*
13. *findDomNodeHandler2() {*
14. *var myDiv = document.getElementById('myDivTwo');*
15. *ReactDOM.findDOMNode(myDivTwo).style.color = 'blue';*
16. *}*
17. *render() {*
18. ***return****(*
19. *<div>*
20. *<h1>ReactJS Find DOM Node Example</h1>*
21. *<button onClick = {****this****.findDomNodeHandler1}>FIND\_DOM\_NODE1</button>*
22. *<button onClick = {****this****.findDomNodeHandler2}>FIND\_DOM\_NODE2</button>*
23. *<h3 id = "myDivOne">JTP-NODE1</h3>*
24. *<h3 id = "myDivTwo">JTP-NODE2</h3>*
25. *</div>*
26. *);*
27. *}*
28. *}*
29. *export****default****App;*

**Output:**



Once you click on the **button**, the color of the node gets changed. It can be shown in the below screen.



# React Forms

Forms are an integral part of any modern web application. It allows the users to interact with the application as well as gather information from the users. Forms can perform many tasks that depend on the nature of your business requirements and logic such as authentication of the user, adding user, searching, filtering, booking, ordering, etc. A form can contain text fields, buttons, checkbox, radio button, etc.

### Creating Form

React offers a stateful, reactive approach to build a form. The component rather than the DOM usually handles the React form. In React, the form is usually implemented by using controlled components.

There are mainly two types of form input in React.

1. Uncontrolled component
2. Controlled component

**Uncontrolled component**

The uncontrolled input is similar to the traditional HTML form inputs. The DOM itself handles the form data. Here, the HTML elements maintain their own state that will be updated when the input value changes. To write an uncontrolled component, you need to use a ref to get form values from the DOM. In other words, there is no need to write an event handler for every state update. You can use a ref to access the input field value of the form from the DOM.

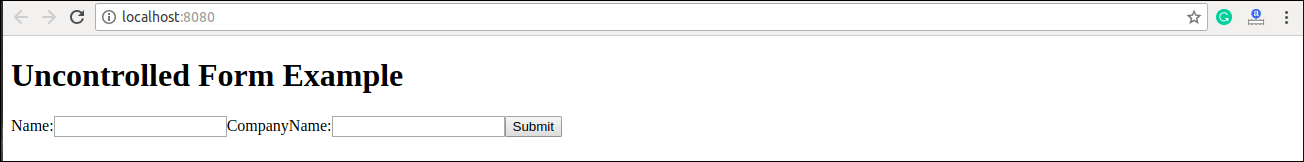
**Example**

In this example, the code accepts a field **username** and **company name** in an uncontrolled component.

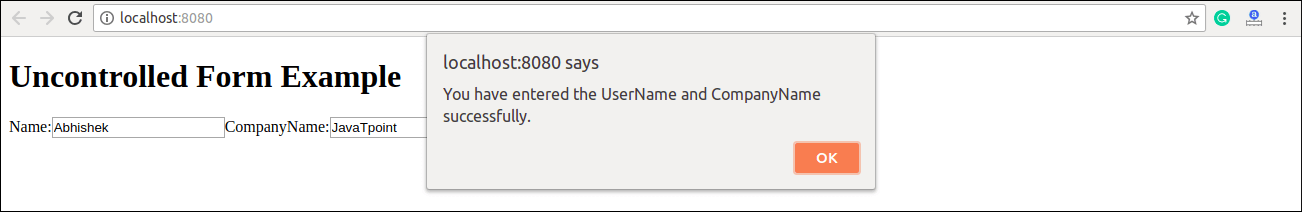
1. ***import****React, { Component } from 'react';*
2. ***class****App****extends****React.Component {*
3. *constructor(props) {*
4. ***super****(props);*
5. ***this****.updateSubmit =****this****.updateSubmit.bind(****this****);*
6. ***this****.input = React.createRef();*
7. *}*
8. *updateSubmit(event) {*
9. *alert('You have entered the UserName and CompanyName successfully.');*
10. *event.preventDefault();*
11. *}*
12. *render() {*
13. ***return****(*
14. *<form onSubmit={****this****.updateSubmit}>*
15. *<h1>Uncontrolled Form Example</h1>*
16. *<label>Name:*
17. *<input type="text" ref={****this****.input} />*
18. *</label>*
19. *<label>*
20. *CompanyName:*
21. *<input type="text" ref={****this****.input} />*
22. *</label>*
23. *<input type="submit" value="Submit" />*
24. *</form>*
25. *);*
26. *}*
27. *}*
28. *export****default****App;*

**Output**

When you execute the above code, you will see the following screen.



After filling the data in the field, you get the message that can be seen in the below screen.



**Controlled Component**

In HTML, form elements typically maintain their own state and update it according to the user input. In the controlled component, the input form element is handled by the component rather than the DOM. Here, the mutable state is kept in the state property and will be updated only with **setState()** method.

Controlled components have functions that govern the data passing into them on every **onChange event**, rather than grabbing the data only once, e.g., when you click a **submit button**. This data is then saved to state and updated with setState() method. This makes component have better control over the form elements and data.

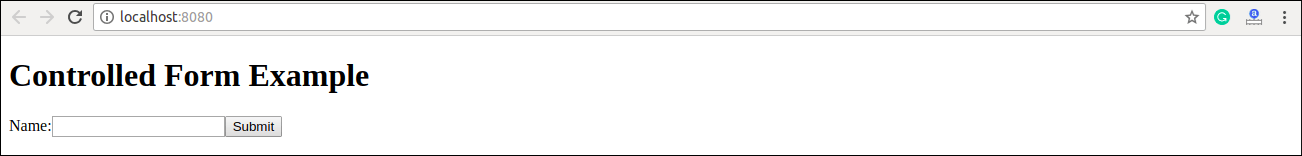
A controlled component takes its current value through **props** and notifies the changes through **callbacks** like an onChange event. A parent component "controls" this changes by handling the callback and managing its own state and then passing the new values as props to the controlled component. It is also called as a "dumb component."

**Example**

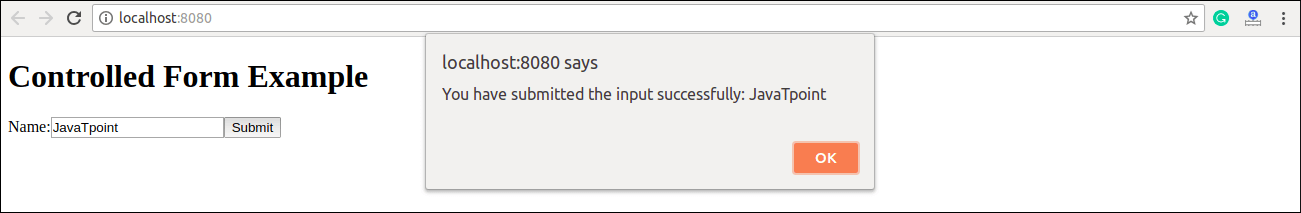
1. ***import****React, { Component } from 'react';*
2. ***class****App****extends****React.Component {*
3. *constructor(props) {*
4. ***super****(props);*
5. ***this****.state = {value: ''};*
6. ***this****.handleChange =****this****.handleChange.bind(****this****);*
7. ***this****.handleSubmit =****this****.handleSubmit.bind(****this****);*
8. *}*
9. *handleChange(event) {*
10. ***this****.setState({value: event.target.value});*
11. *}*
12. *handleSubmit(event) {*
13. *alert('You have submitted the input successfully: ' +****this****.state.value);*
14. *event.preventDefault();*
15. *}*
16. *render() {*
17. ***return****(*
18. *<form onSubmit={****this****.handleSubmit}>*
19. *<h1>Controlled Form Example</h1>*
20. *<label>*
21. *Name:*
22. *<input type="text" value={****this****.state.value} onChange={****this****.handleChange} />*
23. *</label>*
24. *<input type="submit" value="Submit" />*
25. *</form>*
26. *);*
27. *}*
28. *}*
29. *export****default****App;*

Output

When you execute the above code, you will see the following screen.



After filling the data in the field, you get the message that can be seen in the below screen.



### Handling Multiple Inputs in Controlled Component

If you want to handle multiple controlled input elements, add a **name** attribute to each element, and then the handler function decided what to do based on the value of **event.target.name**.

**Example**

1. ***import****React, { Component } from 'react';*
2. ***class****App****extends****React.Component {*
3. *constructor(props) {*
4. ***super****(props);*
5. ***this****.state = {*
6. *personGoing:****true****,*
7. *numberOfPersons: 5*
8. *};*
9. ***this****.handleInputChange =****this****.handleInputChange.bind(****this****);*
10. *}*
11. *handleInputChange(event) {*
12. ***const****target = event.target;*
13. ***const****value = target.type === 'checkbox' ? target.checked : target.value;*
14. ***const****name = target.name;*
15. ***this****.setState({*
16. *[name]: value*
17. *});*
18. *}*
19. *render() {*
20. ***return****(*
21. *<form>*
22. *<h1>Multiple Input Controlled Form Example</h1>*
23. *<label>*
24. *Is Person going:*
25. *<input*
26. *name="personGoing"*
27. *type="checkbox"*
28. *checked={****this****.state.personGoing}*
29. *onChange={****this****.handleInputChange} />*
30. *</label>*
31. *<br />*
32. *<label>*
33. *Number of persons:*
34. *<input*
35. *name="numberOfPersons"*
36. *type="number"*
37. *value={****this****.state.numberOfPersons}*
38. *onChange={****this****.handleInputChange} />*
39. *</label>*
40. *</form>*
41. *);*
42. *}*
43. *}*
44. *export****default****App;*

**Output**



# React Controlled Vs. Uncontrolled Component

### Controlled Component

A controlled component is bound to a value, and its changes will be handled in code by using **event-based callbacks**. Here, the input form element is handled by the react itself rather than the DOM. In this, the mutable state is kept in the state property and will be updated only with setState() method.

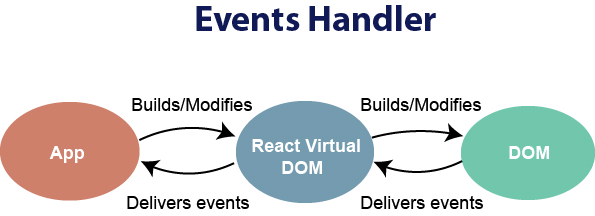
Controlled components have functions that govern the data passing into them on every **onChange** event occurs. This data is then saved to state and updated with setState() method. It makes component have better control over the form elements and data.

### Uncontrolled Component

It is similar to the traditional HTML form inputs. Here, the form data is handled by the DOM itself. It maintains their own state and will be updated when the input value changes. To write an uncontrolled component, there is no need to write an event handler for every state update, and you can use a ref to access the value of the form from the DOM.

## Difference table between controlled and uncontrolled component

|  |  |  |
| --- | --- | --- |
| **SN** | **Controlled** | **Uncontrolled** |
| **1.** | It does not maintain its internal state. | It maintains its internal states. |
| **2.** | Here, data is controlled by the parent component. | Here, data is controlled by the DOM itself. |
| **3.** | It accepts its current value as a prop. | It uses a ref for their current values. |
| **4.** | It allows validation control. | It does not allow validation control. |
| **5.** | It has better control over the form elements and data. | It has limited control over the form elements and data. |



Handling events with react have some syntactic differences from handling events on DOM. These are:

1. React events are named as **camelCase** instead of **lowercase**.
2. With JSX, a function is passed as the **event handler** instead of a **string**. For example:

**Event declaration in plain HTML:**

1. *<button onclick="showMessage()">*
2. *Hello JavaTpoint*
3. *</button>*

**Event declaration in React:**

1. *<button onClick={showMessage}>*
2. *Hello JavaTpoint*
3. *</button>*

3. In react, we cannot return **false** to prevent the **default** behavior. We must call **preventDefault** event explicitly to prevent the default behavior. For example:

In plain HTML, to prevent the default link behavior of opening a new page, we can write:

1. *<a href="#" onclick="console.log('You had clicked a Link.'); return false">*
2. *Click\_Me*
3. *</a>*

In React, we can write it as:

1. *function ActionLink() {*
2. *function handleClick(e) {*
3. *e.preventDefault();*
4. *console.log('You had clicked a Link.');*
5. *}*
6. ***return****(*
7. *<a href="#" onClick={handleClick}>*
8. *Click\_Me*
9. *</a>*
10. *);*
11. *}*

In the above example, e is a **Synthetic Event** which defines according to the **W3C** spec.

Now let us see how to use Event in React.

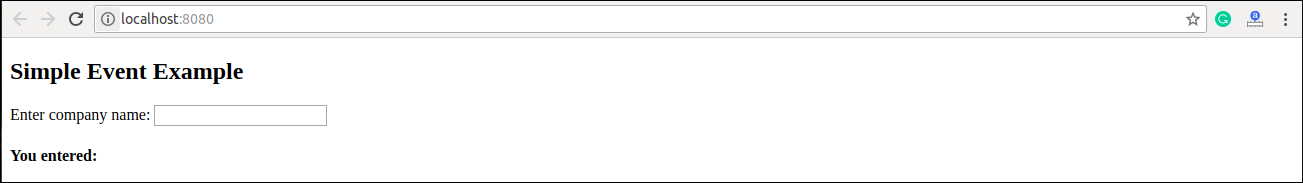
### Example

In the below example, we have used only one component and adding an onChange event. This event will trigger the **changeText** function, which returns the company name.

1. ***import****React, { Component } from 'react';*
2. ***class****App****extends****React.Component {*
3. *constructor(props) {*
4. ***super****(props);*
5. ***this****.state = {*
6. *companyName: ''*
7. *};*
8. *}*
9. *changeText(event) {*
10. ***this****.setState({*
11. *companyName: event.target.value*
12. *});*
13. *}*
14. *render() {*
15. ***return****(*
16. *<div>*
17. *<h2>Simple Event Example</h2>*
18. *<label htmlFor="name">Enter company name: </label>*
19. *<input type="text" id="companyName" onChange={****this****.changeText.bind(****this****)}/>*
20. *<h4>You entered: {****this****.state.companyName }</h4>*
21. *</div>*
22. *);*
23. *}*
24. *}*
25. *export****default****App;*

**Output**

When you execute the above code, you will get the following output.



After entering the name in the textbox, you will get the output as like below screen.



# React Conditional Rendering

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

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

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

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

### if

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

Example

1. *function UserLoggin(props) {*
2. ***return****<h1>Welcome back!</h1>;*
3. *}*
4. *function GuestLoggin(props) {*
5. ***return****<h1>Please sign up.</h1>;*
6. *}*
7. *function SignUp(props) {*
8. ***const****isLoggedIn = props.isLoggedIn;*
9. ***if****(isLoggedIn) {*
10. ***return****<UserLogin />;*
11. *}*
12. ***return****<GuestLogin />;*
13. *}*
15. *ReactDOM.render(*
16. *<SignUp isLoggedIn={****false****} />,*
17. *document.getElementById('root')*
18. *);*

### Logical && operator

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

Syntax:

1. {
2. condition &&
3. // whatever written after && will be a part of output.
4. }

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

If you run the below code, you will not see the **alert** message because the condition is not matching.

1. *('javatpoint' == 'JavaTpoint') && alert('This alert will never be shown!')*

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

1. *(10 > 5) && alert('This alert will be shown!')*

Example

1. ***import****React from 'react';*
2. ***import****ReactDOM from 'react-dom';*
3. *// Example Component*
4. *function Example()*
5. *{*
6. ***return****(<div>*
7. *{*
8. *(10 > 5) && alert('This alert will be shown!')*
9. *}*
10. *</div>*
11. *);*
12. *}*

You can see in the above output that as the condition **(10 > 5)** evaluates to true, the alert message is successfully rendered on the screen

### Ternary operator

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

Syntax

1. *condition ?****true****:****false***

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

Example

1. *render() {*
2. ***const****isLoggedIn =****this****.state.isLoggedIn;*
3. ***return****(*
4. *<div>*
5. *Welcome {isLoggedIn ? 'Back' : 'Please login first'}.*
6. *</div>*
7. *);*
8. *}*

### Switch case operator

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

Example

1. *function NotificationMsg({ text}) {*
2. ***switch****(text) {*
3. ***case****'Hi All':*
4. ***return****<Message: text={text} />;*
5. ***case****'Hello JavaTpoint':*
6. ***return****<Message text={text} />;*
7. ***default****:*
8. ***return******null****;*
9. *}*
10. *}*

### Conditional Rendering with enums

An **enum** is a great way to have a multiple conditional rendering. It is more **readable** as compared to switch case operator. It is perfect for **mapping** between different **state**. It is also perfect for mapping in more than one condition. It can be understood in the below example.

Example

1. *function NotificationMsg({ text, state }) {*
2. ***return****(*
3. *<div>*
4. *{{*
5. *info: <Message text={text} />,*
6. *warning: <Message text={text} />,*
7. *}[state]}*
8. *</div>*
9. *);*
10. *}*

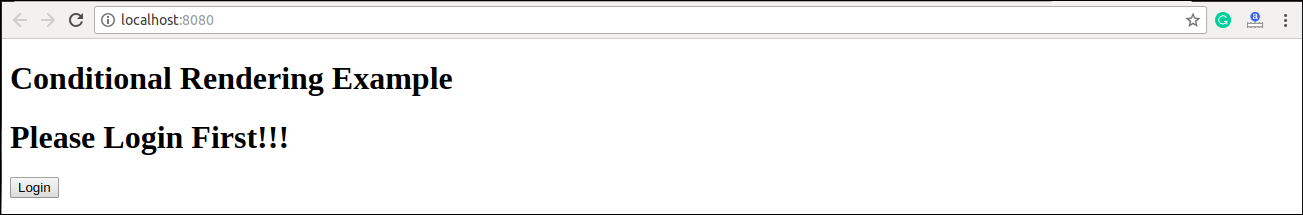
### Conditional Rendering Example

In the below example, we have created a **stateful** component called **App** which maintains the login control. Here, we create three components representing Logout, Login, and Message component. The stateful component App will render either or depending on its current **state**.

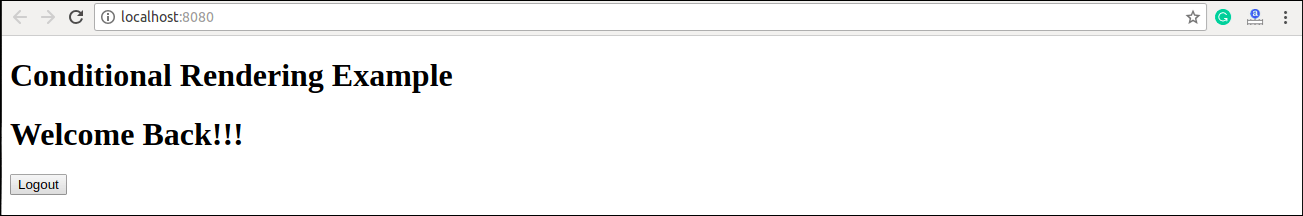
1. ***import****React, { Component } from 'react';*
2. *// Message Component*
3. *function Message(props)*
4. *{*
5. ***if****(props.isLoggedIn)*
6. ***return****<h1>Welcome Back!!!</h1>;*
7. ***else***
8. ***return****<h1>Please Login First!!!</h1>;*
9. *}*
10. *// Login Component*
11. *function Login(props)*
12. *{*
13. ***return****(*
14. *<button onClick = {props.clickInfo}> Login </button>*
15. *);*
16. *}*
17. *// Logout Component*
18. *function Logout(props)*
19. *{*
20. ***return****(*
21. *<button onClick = {props.clickInfo}> Logout </button>*
22. *);*
23. *}*
24. ***class****App****extends****Component{*
25. *constructor(props)*
26. *{*
27. ***super****(props);*
28. ***this****.handleLogin =****this****.handleLogin.bind(****this****);*
29. ***this****.handleLogout =****this****.handleLogout.bind(****this****);*
30. ***this****.state = {isLoggedIn :****false****};*
31. *}*
32. *handleLogin()*
33. *{*
34. ***this****.setState({isLoggedIn :****true****});*
35. *}*
36. *handleLogout()*
37. *{*
38. ***this****.setState({isLoggedIn :****false****});*
39. *}*
40. *render(){*
41. ***return****(*
42. *<div>*
43. *<h1> Conditional Rendering Example </h1>*
44. *<Message isLoggedIn = {****this****.state.isLoggedIn}/>*
45. *{*
46. *(****this****.state.isLoggedIn)?(*
47. *<Logout clickInfo = {****this****.handleLogout} />*
48. *) : (*
49. *<Login clickInfo = {****this****.handleLogin} />*
50. *)*
51. *}*
52. *</div>*
53. *);*
54. *}*
55. *}*
56. *export****default****App;*

**Output:**

When you execute the above code, you will get the following screen.



After clicking the logout button, you will get the below screen.



### Preventing Component form Rendering

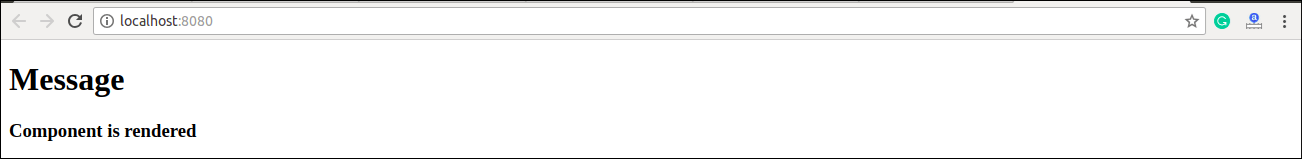
Sometimes it might happen that a component hides itself even though another component rendered it. To do this (prevent a component from rendering), we will have to return **null** instead of its render output. It can be understood in the below example:

Example

In this example, the is rendered based on the value of the prop called **displayMessage**. If the prop value is false, then the component does not render.

1. ***import****React from 'react';*
2. ***import****ReactDOM from 'react-dom';*
3. *function Show(props)*
4. *{*
5. ***if****(!props.displayMessage)*
6. ***return******null****;*
7. ***else***
8. ***return****<h3>Component is rendered</h3>;*
9. *}*
10. *ReactDOM.render(*
11. *<div>*
12. *<h1>Message</h1>*
13. *<Show displayMessage = {****true****} />*
14. *</div>,*
15. *document.getElementById('app')*
16. *);*

**Output:**



### React Lists

Lists are used to display data in an ordered format and mainly used to display menus on websites. In React, Lists can be created in a similar way as we create lists in JavaScript. Let us see how we transform Lists in regular JavaScript.

The map() function is used for traversing the lists. In the below example, the map() function takes an array of numbers and multiply their values with 5. We assign the new array returned by map() to the variable multiplyNums and log it.

Example

1. *var numbers = [1, 2, 3, 4, 5];*
2. ***const****multiplyNums = numbers.map((number)=>{*
3. ***return****(number \* 5);*
4. *});*
5. *console.log(multiplyNums);*

**Output**

The above JavaScript code will log the output on the console. The output of the code is given below.

[5, 10, 15, 20, 25]

Now, let us see how we create a list in React. To do this, we will use the map() function for traversing the list element, and for updates, we enclosed them between **curly braces {}**. Finally, we assign the array elements to listItems. Now, include this new list inside **<ul> </ul>** elements and render it to the DOM.

Example

1. ***import****React from 'react';*
2. ***import****ReactDOM from 'react-dom';*
4. ***const****myList = ['Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa'];*
5. ***const****listItems = myList.map((myList)=>{*
6. ***return****<li>{myList}</li>;*
7. *});*
8. *ReactDOM.render(*
9. *<ul> {listItems} </ul>,*
10. *document.getElementById('app')*
11. *);*
12. *export****default****App;*

**Output**

## React Lists

## Rendering Lists inside components

In the previous example, we had directly rendered the list to the DOM. But it is not a good practice to render lists in React. In React, we had already seen that everything is built as individual components. Hence, we would need to render lists inside a component. We can understand it in the following code.

Example

1. ***import****React from 'react';*
2. ***import****ReactDOM from 'react-dom';*
4. *function NameList(props) {*
5. ***const****myLists = props.myLists;*
6. ***const****listItems = myLists.map((myList) =>*
7. *<li>{myList}</li>*
8. *);*
9. ***return****(*
10. *<div>*
11. *<h2>Rendering Lists inside component</h2>*
12. *<ul>{listItems}</ul>*
13. *</div>*
14. *);*
15. *}*
16. ***const****myLists = ['Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa'];*
17. *ReactDOM.render(*
18. *<NameList myLists={myLists} />,*
19. *document.getElementById('app')*
20. *);*
21. *export****default****App;*

**Output**

