**ReactJS Tutorial**

ReactJS tutorial provides basic and advanced concepts of ReactJS. Currently, ReactJS is one of the most popular JavaScript front-end libraries which has a strong foundation and a large community.

ReactJS is a **declarative**, **efficient**, and flexible **JavaScript library** for building reusable UI components. It is an open-source, component-based front end library which is responsible only for the view layer of the application. It was initially developed and maintained by Facebook and later used in its products like WhatsApp & Instagram.

Our ReactJS tutorial includes all the topics which help to learn ReactJS. These are ReactJS Introduction, ReactJS Features, ReactJS Installation, Pros and Cons of ReactJS, ReactJS JSX, ReactJS Components, ReactJS State, ReactJS Props, ReactJS Forms, ReactJS Events, ReactJS Animation and many more.

Why we use ReactJS?

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

# **React Introduction**

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

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

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.

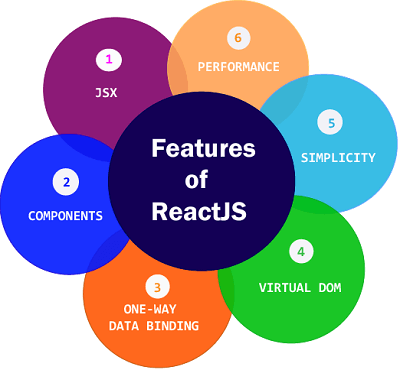
To create React app, we write React components that correspond to various elements. We organize these components inside higher level components which define the application structure. For example, we take a form that consists of many elements like input fields, labels, or buttons. We can write each element 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 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 Native

React Native is an open-source JavaScript framework used for developing a mobile application for iOS Android, and Windows. It uses only JavaScript to build a cross-platform mobile app. React Native is same as React, but it uses native components instead of using web components as building blocks. It targets mobile platforms rather than the browser.

Facebook develops the React Native in 2013 for its internal project Hackathon. In March 2015, Facebook announced that React Native is open and available on GitHub.

React Native was initially developed for the iOS application. However, recently, it also supports the Android operating system.

### **Advantages of React Native**

There are several advantages of React Native for building mobile applications. Some of them are given below:

1. **Cross-Platform Usage:** It provides the facility of "Learn once write everywhere." It works for both platform Android as well as iOS devices.
2. **Class Performance:** The code written in React Native are compiled into native code, which enables it for both operating systems as well as it functions in the same way on both the platforms.
3. **JavaScript:** JavaScript knowledge is used to build native mobile apps.
4. **Community:** The large community of ReactJS and React Native helps us to find any answer we require.
5. **Hot Reloading:** Making a few changes in the code of your app will be immediately visible during development. If the business logic is changed, its reflection is live reloaded on screen.
6. **Improving with Time:** Some features of iOS and Android are still not supported, and the community is always inventing the best practices.
7. **Native Components:** We will need to write some platform specific code if we want to create native functionality, which is not designed yet.
8. **Existence is Uncertain:** As Facebook develop this framework, its presence is uncertain since it keeps all the rights to kill off the project anytime. As the popularity of React Native rises, it is unlikely to happen.

### **Disadvantage of React Native**

1. **React Native is Still New and Immature:** React Native is a newbie in Android and iOS programming languages and is still in its improvement stage, which can have a negative impact on the apps.
2. **Learning is Tough:** React Native is not easy to learn, especially for a fresher in the app development field.
3. **It Lacks the Security Robustness:** React Native is a JavaScript library and open-source framework, which creates a gap in the security robustness. When you are creating banking and financial apps where data is highly confidential, experts advice not to choose React Native.
4. **It Takes More Time to Initialize:** React Native takes a lot of time for initializing the runtime even for the hi-tech gadgets and devices.

# **React vs. Vue**

React and Vue is the two most popular JavaScript libraries which are used to build thousands of websites today. Both React and Vue are very powerful frameworks with their own set of pros and cons. Which one you have to pick, depends on the business needs and use cases.

Both React and Vue have a lot of common things like the component-based architecture, usage of virtual DOM, usage of props, chrome Dev tools for debugging, and many more. But, both have some significant differences, which are given below.

# **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**

1. <div>Hello JavaTpoint</div>

**Corresponding Output**

1. 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**

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

**class** App **extends** Component{

   render(){

**return**(

         <div>

            <h1>JavaTpoint</h1>

          <h2>Training Institutes</h2>

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

         </div>

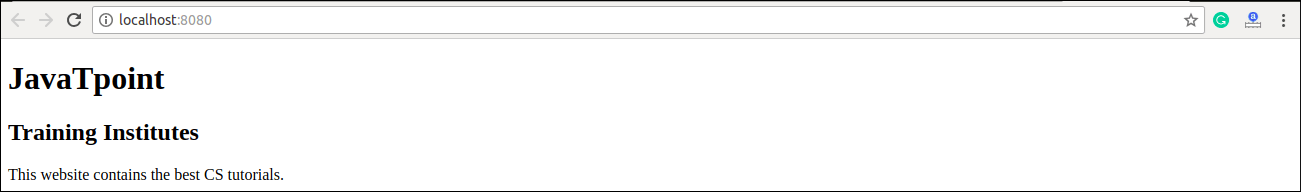
      );

   }

}

export **default** App;

**Output:**



JSX Attributes

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

### **Example**

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

**class** App **extends** Component{

   render(){

**return**(

         <div>

             <h1>JavaTpoint</h1>

           <h2>Training Institutes</h2>

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

         </div>

      );

   }

}

export **default** App;

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

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

1. var element = <h2 className = "firstAttribute">Hello JavaTpoint</h2

**Example:**

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

**class** App **extends** Component{

   render(){

**return**(

         <div>

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

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

         </div>

      );

   }

}

export **default** App;

**Output:**

JavaTpoint

This website contains the CS best tutorials.

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

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

**Example**

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

**class** App **extends** Component{

   render(){

**return**(

         <div>

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

         </div>

      );

   }

}

export **default** App;

**Output:**

**45**

## JSX Comments

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

## JSX Styling

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

**Example:**

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

**class** App **extends** Component{

   render(){

     var myStyle = {

         fontSize: 80,

         fontFamily: 'Courier',

         color: '#003300'

      }

**return** (

         <div>

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

         </div>

      );

   }

}

export **default** App;

**Output:**

[**www.javatpoint.com**](http://www.javatpoint.com)

**Note:**

**JSX** cannot allow to use if-else statements. Instead of it, you can use conditional (ternary) expressions.

# **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.

function WelcomeMessage(props) {

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

}

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

**Example**:

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

**class** App **extends** React.Component {

   render() {

**return** (

         <div>

            <First/>

            <Second/>

         </div>

      );

   }

}

**class** First **extends** React.Component {

   render() {

**return** (

       <div>

            <h1>JavaTpoint</h1>

         </div>

      );

   }

}

**class** Second **extends** React.Component {

   render() {

**return** (

         <div>

            <h2>www.javatpoint.com</h2>

            <p>This websites contains the great CS tutorial.</p>

         </div>

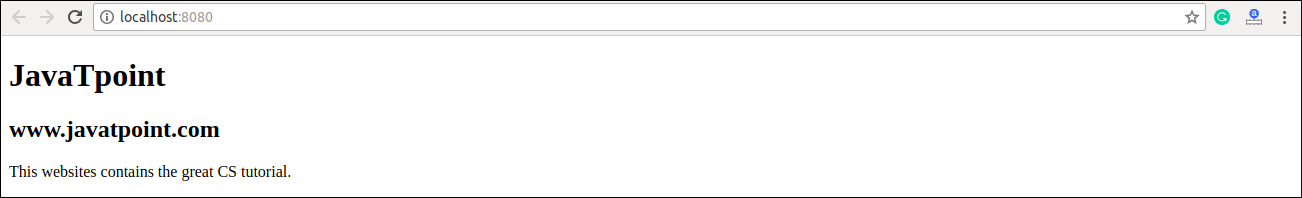
      );

   }

}

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.

**class** MyComponent **extends** React.Component {

  render() {

**return** (

      <div>This is main component.</div>

    );

  }

}

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.

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

**class** App **extends** React.Component {

 constructor() {

**super**();

**this**.state = {

         data:

         [

            {

               "name":"Abhishek"

            },

            {

               "name":"Saharsh"

            },

            {

               "name":"Ajay"

            }

         ]

      }

   }

   render() {

**return** (

         <div>

            <StudentName/>

            <ul>

                {**this**.state.data.map((item) => <List data = {item} />)}

            </ul>

         </div>

      );

   }

}

**class** StudentName **extends** React.Component {

   render() {

**return** (

         <div>

            <h1>Student Name Detail</h1>

         </div>

      );

   }

}

**class** List **extends** React.Component {

   render() {

**return** (

         <ul>

            <li>{**this**.props.data.name}</li>

         </ul>

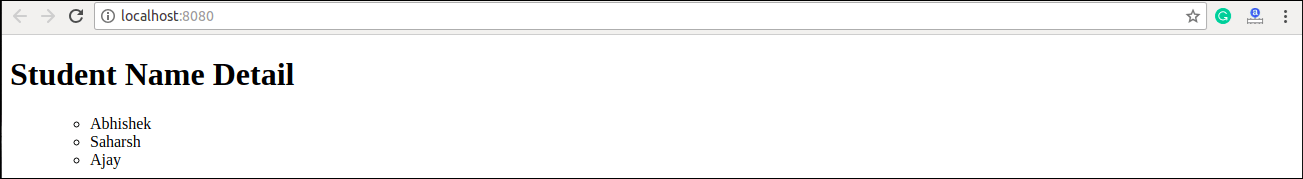
      );

   }

}

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.

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

**class** App **extends** React.Component {

 constructor() {

**super**();

**this**.state = { displayBio: **true** };

      }

      render() {

**const** bio = **this**.state.displayBio ? (

              <div>

                  <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>

            </div>

              ) : **null**;

**return** (

                  <div>

                      <h1> Welcome to JavaTpoint!! </h1>

                      { bio }

                  </div>

              );

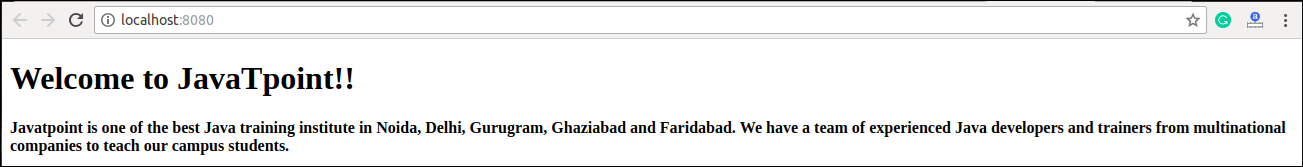
     }

}

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.

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

**class** App **extends** React.Component {

 constructor() {

**super**();

**this**.state = { displayBio: **false** };

      console.log('Component this', **this**);

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

      }

      toggleDisplayBio(){

**this**.setState({displayBio: !**this**.state.displayBio});

          }

      render() {

**return** (

              <div>

                  <h1>Welcome to JavaTpoint!!</h1>

                  {

**this**.state.displayBio ? (

                          <div>

                              <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>

                              <button onClick={**this**.toggleDisplayBio}> Show Less </button>

                        </div>

                          ) : (

                              <div>

                               <button onClick={**this**.toggleDisplayBio}> Read More </button>

                              </div>

                          )

                  }

             </div>

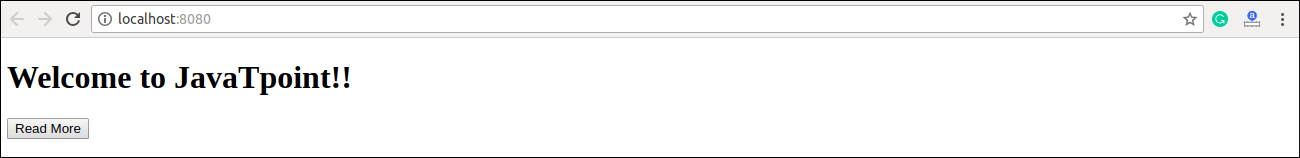
        )

    }

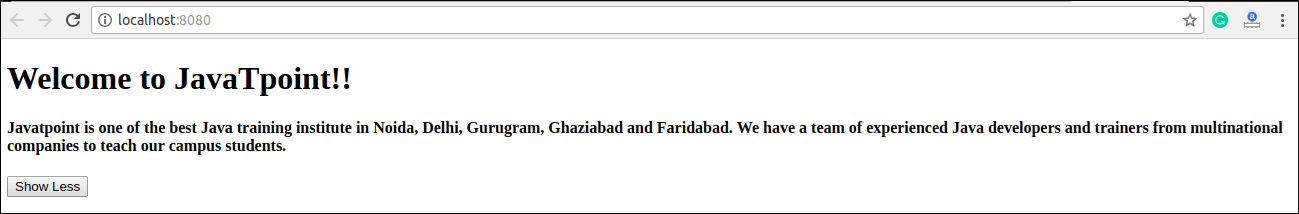
}

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**

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

**class** App **extends** React.Component {

   render() {

**return** (

          <div>

            <h1> Welcome to { **this**.props.name } </h1>

            <p> <h4> Javatpoint is one of the best Java training institute in Noida, Delhi, Gurugram, Ghaziabad and Faridabad. </h4> </p>

          </div>

      );

   }

}

export **default** App;

**Main.js**

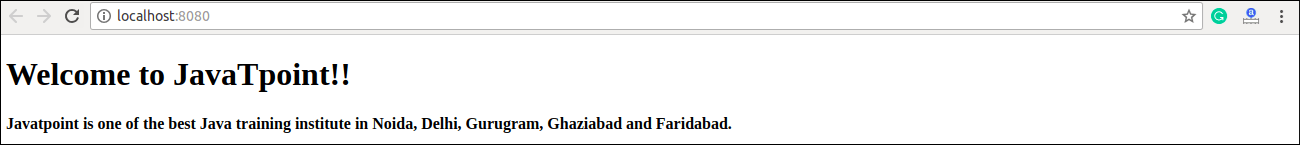
**import** React from 'react';

**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**

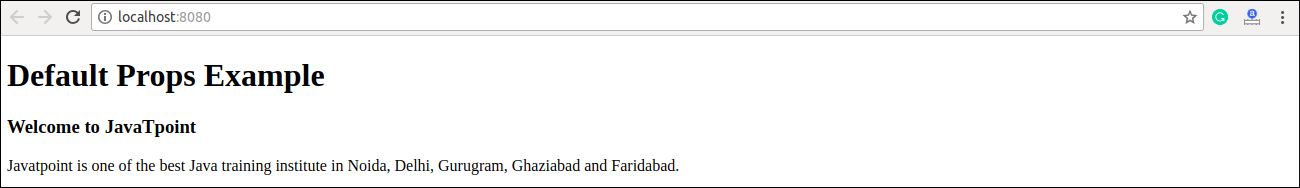
**App.js**

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**

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

**class** App **extends** React.Component {

   constructor(props) {

**super**(props);

**this**.state = {

         name: "JavaTpoint",

      }

   }

   render() {

**return** (

         <div>

            <JTP jtpProp = {**this**.state.name}/>

         </div>

      );

   }

}

**class** JTP **extends** React.Component {

   render() {

**return** (

          <div>

              <h1>State & Props Example</h1>

              <h3>Welcome to {**this**.props.jtpProp}</h3>

            <p>Javatpoint is one of the best Java training institute in Noida, Delhi,Gurugram, Ghaziabad and Faridabad.</p>

         </div>

      );

   }

}

export **default** App;

**Main.js**

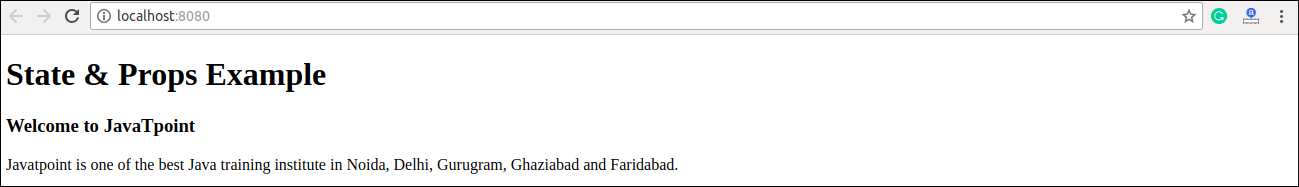
**import** React from 'react';

**import** ReactDOM from 'react-dom';

**import** App from './App.js';

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

**Output:**



# **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 \*/};

### **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**

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

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

**class** App **extends** React.Component {

   render() {

**return** (

          <div>

              <h1>ReactJS Props validation example</h1>

              <table>

                  <tr>

                      <th>Type</th>

                      <th>Value</th>

                      <th>Valid</th>

                  </tr>

                <tr>

                      <td>Array</td>

                      <td>{**this**.props.propArray}</td>

                      <td>{**this**.props.propArray ? "true" : "False"}</td>

                  </tr>

                  <tr>

                      <td>Boolean</td>

                      <td>{**this**.props.propBool ? "true" : "False"}</td>

                      <td>{**this**.props.propBool ? "true" : "False"}</td>

                  </tr>

                  <tr>

                      <td>Function</td>

                      <td>{**this**.props.propFunc(5)}</td>

                      <td>{**this**.props.propFunc(5) ? "true" : "False"}</td>

                  </tr>

                  <tr>

                      <td>String</td>

                      <td>{**this**.props.propString}</td>

                      <td>{**this**.props.propString ? "true" : "False"}</td>

                  </tr>

                  <tr>

                      <td>Number</td>

                      <td>{**this**.props.propNumber}</td>

                      <td>{**this**.props.propNumber ? "true" : "False"}</td>

                  </tr>

             </table>

        </div>

        );

   }

}

App.propTypes = {

    propArray: PropTypes.array.isRequired,

    propBool: PropTypes.bool.isRequired,

    propFunc: PropTypes.func,

    propNumber: PropTypes.number,

    propString: PropTypes.string,

}

App.defaultProps = {

    propArray: [1,2,3,4,5],

    propBool: **true**,

    propFunc: function(x){**return** x+5},

    propNumber: 1,

    propString: "JavaTpoint",

}

export **default** App;

**Main.js**

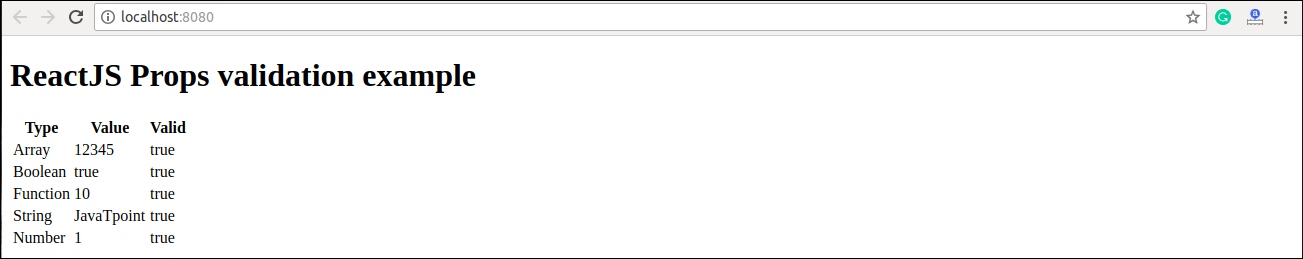
**import** React from 'react';

**import** ReactDOM from 'react-dom';

**import** App from './App.js';

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

**Output:**



## 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**

var Component = React.createClass({

App.propTypes = {

   customProp: function(props, propName, componentName) {

**if** (!item.isValid(props[propName])) {

**return** **new** Error('Validation failed!');

        }

      }

   }

})

# **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. |

The below table will guide you about the changing in props and state.

|  |  |  |  |
| --- | --- | --- | --- |
| **SN** | **Condition** | **Props** | **State** |
| **1.** | Can get initial value from parent Component? | Yes | Yes |
| **2.** | Can be changed by parent Component? | Yes | No |
| **3.** | Can set default values inside Component? | Yes | Yes |
| **4.** | Can change inside Component? | No | Yes |
| **5.** | Can set initial value for child Components? | Yes | Yes |
| **6.** | Can change in child Components? | Yes | No |

The component state and props share some common qualities . They are:

1. Both are plain JS Object.
2. Both can contain default values.
3. Both are read-only when they are using by this.

# **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. }

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

  }

  handleEvent(){

    console.log(**this**.props);

  }

  render() {

**return** (

      <div className="App">

    <h2>React Constructor Example</h2>

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

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

      </div>

    );

  }

}

export **default** App;

**Main.js**

**import** React from 'react';

**import** ReactDOM from 'react-dom';

**import** App from './App.js';

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

**Output**

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.

**class** App **extends** Component {

    render () {

**return** (

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

        );

    }

}

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

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

**class** App **extends** Component {

    constructor(props){

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

    }

    render () {

**return** (

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

        );

    }

}

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

## Arrow Functions

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

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

**class** App **extends** Component {

  constructor(props){

**super**(props);

**this**.state = {

         data: 'www.javatpoint.com'

      }

  }

  handleEvent = () => {

    console.log(**this**.props);

  }

  render() {

**return** (

      <div className="App">

    <h2>React Constructor Example</h2>

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

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

      </div>

    );

  }

}

export **default** App;

We can use a constructor in the following ways:

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

**class** App **extends** Component {

  constructor(props){

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

**this**.state = {

            inputTextValue: 'initial value',

        };

  }

}

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

**class** App **extends** Component {

    constructor(props) {

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

**super**();

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

**super**(props);

    }

}

**3) Initializing third-party libraries**

**class** App **extends** Component {

    constructor(props) {

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

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

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

    }

}

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

**class** App **extends** Component {

    constructor(props) {

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

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

    }

}

# **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**

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

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

**class** App **extends** React.Component {

   constructor() {

**super**();

**this**.state = {

          msg: "Welcome to JavaTpoint"

      };

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

   }

   updateSetState() {

**this**.setState({

          msg:"Its a best ReactJS tutorial"

       });

   }

   render() {

**return** (

         <div>

             <h1>{**this**.state.msg}</h1>

             <button onClick = {**this**.updateSetState}>SET STATE</button>

         </div>

      );

   }

}

export **default** App;

**Main.js**

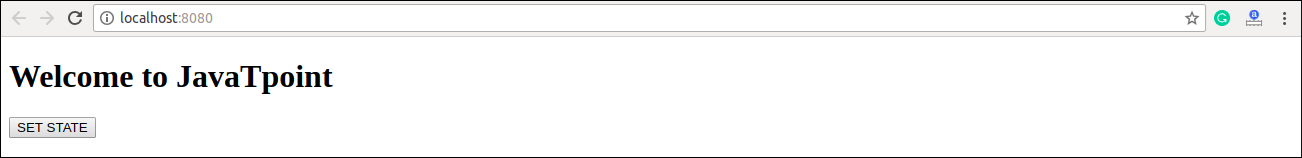
**import** React from 'react';

**import** ReactDOM from 'react-dom';

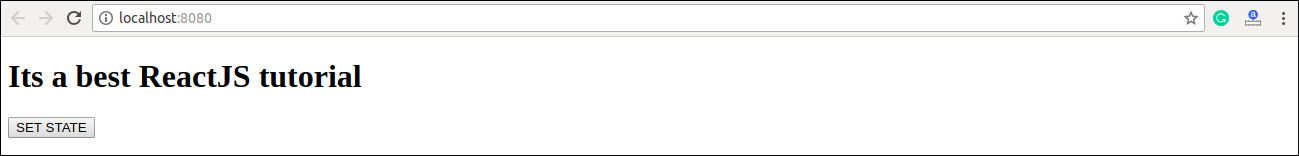
**import** App from './App.js';

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

**Output:**



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**

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

**class** App **extends** React.Component {

   constructor() {

**super**();

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

   }

   forceUpdateState() {

**this**.forceUpdate();

   };

   render() {

**return** (

         <div>

             <h1>Example to generate random number</h1>

             <h3>Random number: {Math.random()}</h3>

             <button onClick = {**this**.forceUpdateState}>ForceUpdate</button>

         </div>

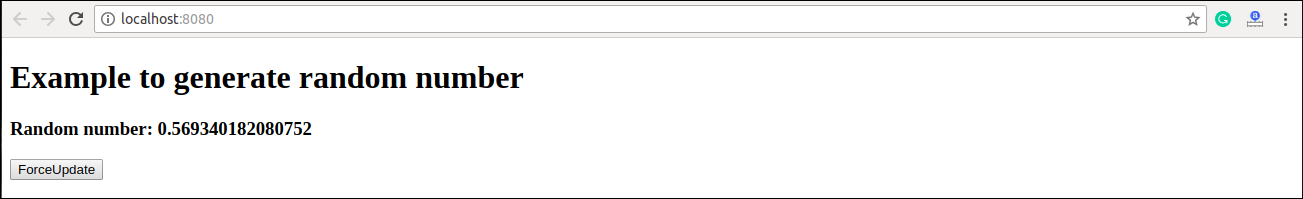
      );

   }

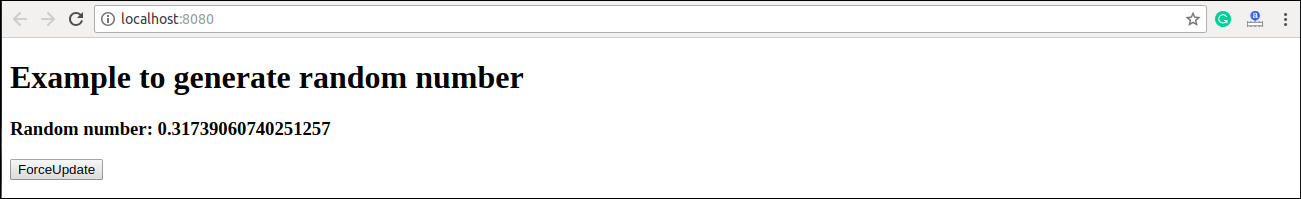
}

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**

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

**import** ReactDOM from 'react-dom';

**class** App **extends** React.Component {

   constructor() {

**super**();

**this**.findDomNodeHandler1 = **this**.findDomNodeHandler1.bind(**this**);

**this**.findDomNodeHandler2 = **this**.findDomNodeHandler2.bind(**this**);

   };

   findDomNodeHandler1() {

       var myDiv = document.getElementById('myDivOne');

      ReactDOM.findDOMNode(myDivOne).style.color = 'red';

   }

   findDomNodeHandler2() {

       var myDiv = document.getElementById('myDivTwo');

       ReactDOM.findDOMNode(myDivTwo).style.color = 'blue';

   }

   render() {

**return** (

         <div>

             <h1>ReactJS Find DOM Node Example</h1>

             <button onClick = {**this**.findDomNodeHandler1}>FIND\_DOM\_NODE1</button>

        <button onClick = {**this**.findDomNodeHandler2}>FIND\_DOM\_NODE2</button>

             <h3 id = "myDivOne">JTP-NODE1</h3>

             <h3 id = "myDivTwo">JTP-NODE2</h3>

         </div>

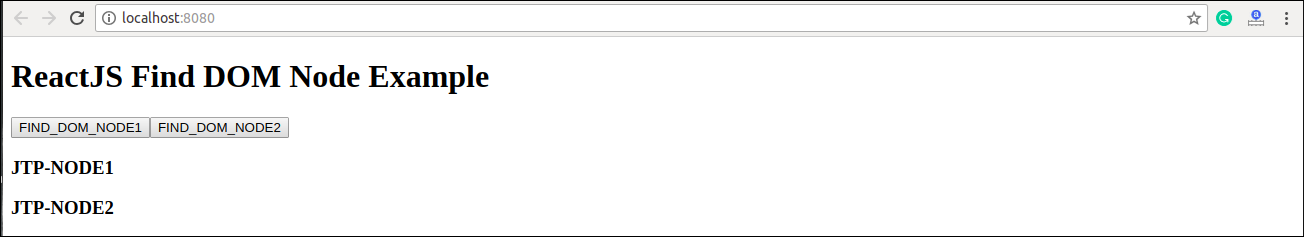
      );

   }

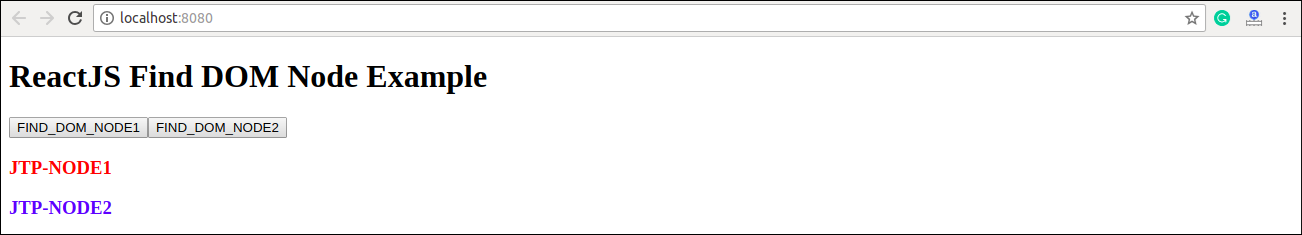
}

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 Component Life-Cycle**

In ReactJS, every component creation process involves various lifecycle methods. These lifecycle methods are termed as component's lifecycle. These lifecycle methods are not very complicated and called at various points during a component's life. The lifecycle of the component is divided into **four phases**. They are:

1. Initial Phase
2. Mounting Phase
3. Updating Phase
4. Unmounting Phase

Each phase contains some lifecycle methods that are specific to the particular phase. Let us discuss each of these phases one by one.

## 1. Initial Phase

It is the **birth** phase of the lifecycle of a ReactJS component. Here, the component starts its journey on a way to the DOM. In this phase, a component contains the default Props and initial State. These default properties are done in the constructor of a component. The initial phase only occurs once and consists of the following methods.

* **getDefaultProps()**  
  It is used to specify the default value of this.props. It is invoked before the creation of the component or any props from the parent is passed into it.
* **getInitialState()**  
  It is used to specify the default value of this.state. It is invoked before the creation of the component.

## 2. Mounting Phase

In this phase, the instance of a component is created and inserted into the DOM. It consists of the following methods.

* **componentWillMount()**  
  This is invoked immediately before a component gets rendered into the DOM. In the case, when you call **setState()** inside this method, the component will not **re-render**.
* **componentDidMount()**  
  This is invoked immediately after a component gets rendered and placed on the DOM. Now, you can do any DOM querying operations.
* **render()**  
  This method is defined in each and every component. It is responsible for returning a single root **HTML node** element. If you don't want to render anything, you can return a **null** or **false** value.

## 3. Updating Phase

It is the next phase of the lifecycle of a react component. Here, we get new **Props** and change **State**. This phase also allows to handle user interaction and provide communication with the components hierarchy. The main aim of this phase is to ensure that the component is displaying the latest version of itself. Unlike the Birth or Death phase, this phase repeats again and again. This phase consists of the following methods.

* **componentWillRecieveProps()**  
  It is invoked when a component receives new props. If you want to update the state in response to prop changes, you should compare this.props and nextProps to perform state transition by using **this.setState()** method.
* **shouldComponentUpdate()**  
  It is invoked when a component decides any changes/updation to the DOM. It allows you to control the component's behavior of updating itself. If this method returns true, the component will update. Otherwise, the component will skip the updating.
* **componentWillUpdate()**  
  It is invoked just before the component updating occurs. Here, you can't change the component state by invoking **this.setState()** method. It will not be called, if **shouldComponentUpdate()** returns false.
* **render()**  
  It is invoked to examine **this.props** and **this.state** and return one of the following types: React elements, Arrays and fragments, Booleans or null, String and Number. If shouldComponentUpdate() returns false, the code inside render() will be invoked again to ensure that the component displays itself properly.
* **componentDidUpdate()**  
  It is invoked immediately after the component updating occurs. In this method, you can put any code inside this which you want to execute once the updating occurs. This method is not invoked for the initial render.

## 4. Unmounting Phase

It is the final phase of the react component lifecycle. It is called when a component instance is **destroyed** and **unmounted** from the DOM. This phase contains only one method and is given below.

* **componentWillUnmount()**  
  This method is invoked immediately before a component is destroyed and unmounted permanently. It performs any necessary **cleanup** related task such as invalidating timers, event listener, canceling network requests, or cleaning up DOM elements. If a component instance is unmounted, you cannot mount it again.

### **Example**

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

**class** App **extends** React.Component {

   constructor(props) {

**super**(props);

**this**.state = {hello: "JavaTpoint"};

**this**.changeState = **this**.changeState.bind(**this**)

   }

   render() {

**return** (

        <div>

             <h1>ReactJS component's Lifecycle</h1>

             <h3>Hello {**this**.state.hello}</h3>

             <button onClick = {**this**.changeState}>Click Here!</button>

         </div>

      );

   }

   componentWillMount() {

      console.log('Component Will MOUNT!')

   }

   componentDidMount() {

      console.log('Component Did MOUNT!')

   }

   changeState(){

**this**.setState({hello:"All!!- Its a great reactjs tutorial."});

   }

   componentWillReceiveProps(newProps) {

      console.log('Component Will Recieve Props!')

   }

   shouldComponentUpdate(newProps, newState) {

**return** **true**;

   }

   componentWillUpdate(nextProps, nextState) {

      console.log('Component Will UPDATE!');

   }

   componentDidUpdate(prevProps, prevState) {

      console.log('Component Did UPDATE!')

   }

   componentWillUnmount() {

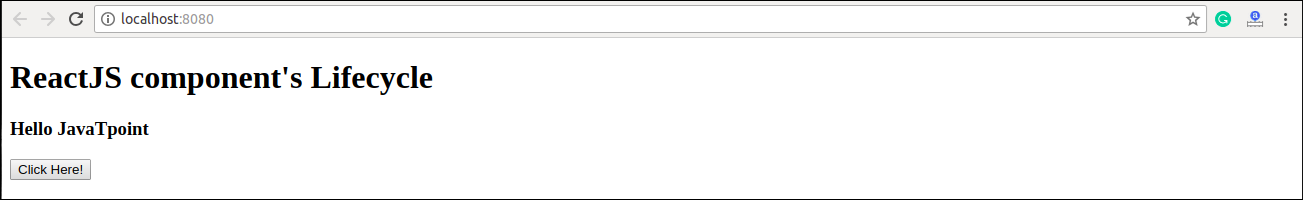
      console.log('Component Will UNMOUNT!')

   }

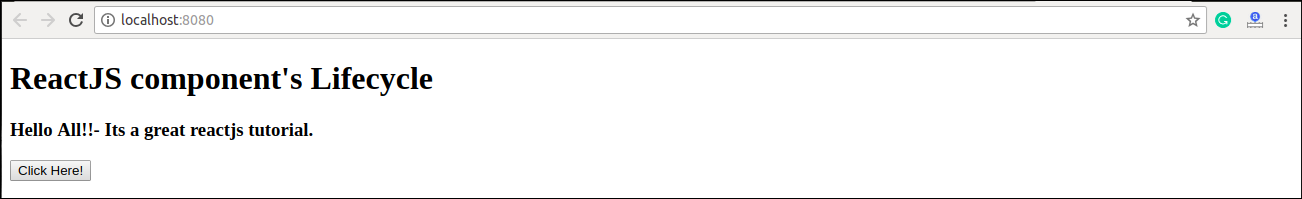
}

export **default** App;

**Output:**



When you click on the **Click Here** Button, you get the updated result which is 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.

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

**class** App **extends** React.Component {

  constructor(props) {

**super**(props);

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

**this**.input = React.createRef();

  }

  updateSubmit(event) {

      alert('You have entered the UserName and CompanyName successfully.');

      event.preventDefault();

  }

  render() {

**return** (

      <form onSubmit={**this**.updateSubmit}>

        <h1>Uncontrolled Form Example</h1>

        <label>Name:

            <input type="text" ref={**this**.input} />

        </label>

      <label>

            CompanyName:

            <input type="text" ref={**this**.input} />

        </label>

        <input type="submit" value="Submit" />

      </form>

    );

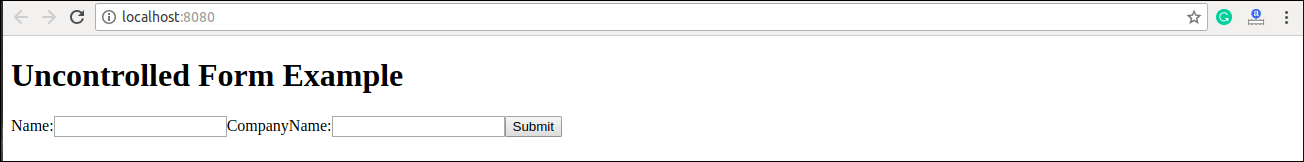
  }

}

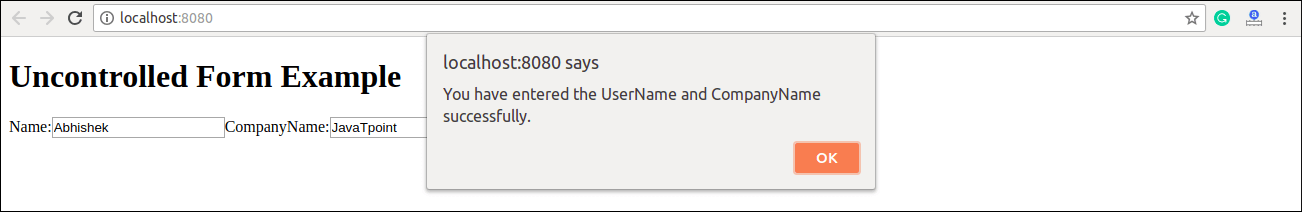
export **default** App;

**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**

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

**class** App **extends** React.Component {

  constructor(props) {

**super**(props);

**this**.state = {value: ''};

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

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

  }

  handleChange(event) {

**this**.setState({value: event.target.value});

  }

  handleSubmit(event) {

      alert('You have submitted the input successfully: ' + **this**.state.value);

      event.preventDefault();

  }

  render() {

**return** (

          <form onSubmit={**this**.handleSubmit}>

            <h1>Controlled Form Example</h1>

            <label>

                Name:

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

            </label>

            <input type="submit" value="Submit" />

         </form>

      );

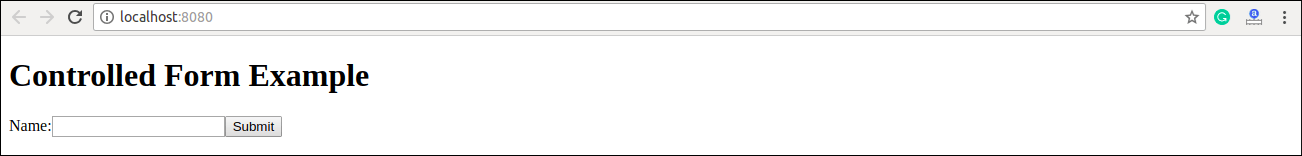
  }

}

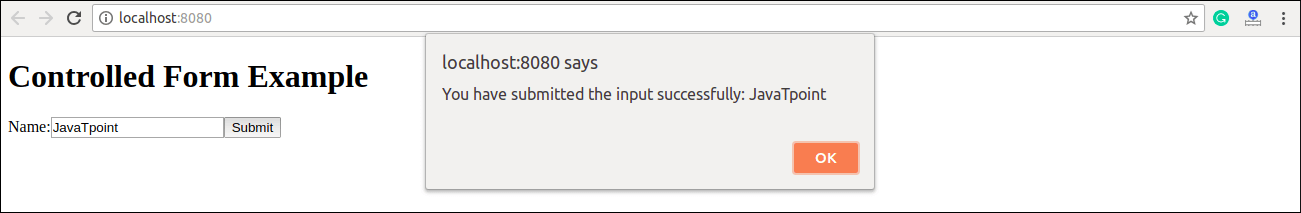
export **default** App;

**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**

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

**class** App **extends** React.Component {

    constructor(props) {

**super**(props);

**this**.state = {

            personGoing: **true**,

            numberOfPersons: 5

        };

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

   }

   handleInputChange(event) {

**const** target = event.target;

**const** value = target.type === 'checkbox' ? target.checked : target.value;

**const** name = target.name;

**this**.setState({

           [name]: value

       });

  }

  render() {

**return** (

          <form>

              <h1>Multiple Input Controlled Form Example</h1>

              <label>

                  Is Person going:

                  <input

                    name="personGoing"

                    type="checkbox"

                   checked={**this**.state.personGoing}

                    onChange={**this**.handleInputChange} />

             </label>

             <br />

             <label>

                 Number of persons:

                 <input

                 name="numberOfPersons"

                 type="number"

                 value={**this**.state.numberOfPersons}

                 onChange={**this**.handleInputChange} />

             </label>

         </form>

     );

  }

}

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.

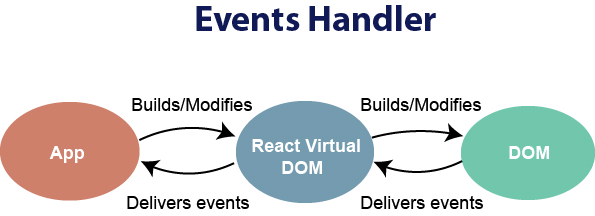
## Difference table between controlled and uncontrolled component

|  |  |  |
| --- | --- | --- |
| **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. |

# **React Events**

An event is an action that could be triggered as a result of the user action or system generated event. For example, a mouse click, loading of a web page, pressing a key, window resizes, and other interactions are called events.

React has its own event handling system which is very similar to handling events on DOM elements. The react event handling system is known as Synthetic Events. The synthetic event is a cross-browser wrapper of the browser's native event.



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:**

<button onclick="showMessage()">

       Hello JavaTpoint

</button>

**Event declaration in React:**

<button onClick={showMessage}>

      Hello JavaTpoint

</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:

<a href="#" onclick="console.log('You had clicked a Link.'); return false">

    Click\_Me

</a>

In React, we can write it as:

function ActionLink() {

    function handleClick(e) {

        e.preventDefault();

        console.log('You had clicked a Link.');

    }

**return** (

        <a href="#" onClick={handleClick}>

              Click\_Me

        </a>

    );

}

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.

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

**class** App **extends** React.Component {

    constructor(props) {

**super**(props);

**this**.state = {

            companyName: ''

        };

    }

    changeText(event) {

**this**.setState({

            companyName: event.target.value

        });

    }

    render() {

**return** (

            <div>

                <h2>Simple Event Example</h2>

                <label htmlFor="name">Enter company name: </label>

     <input type="text" id="companyName" onChange={**this**.changeText.bind(**this**)}/>

                <h4>You entered: { **this**.state.companyName }</h4>

            </div>

        );

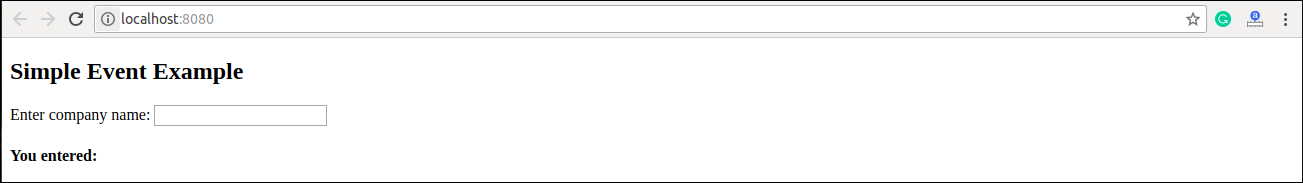
    }

}

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**

function UserLoggin(props) {

**return** <h1>Welcome back!</h1>;

}

function GuestLoggin(props) {

**return** <h1>Please sign up.</h1>;

}

function SignUp(props) {

**const** isLoggedIn = props.isLoggedIn;

**if** (isLoggedIn) {

**return** <UserLogin />;

  }

**return** <GuestLogin />;

}

ReactDOM.render(

  <SignUp isLoggedIn={**false**} />,

  document.getElementById('root')

);

## Logical && operator

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

### **Syntax**

{

    condition &&

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

}

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

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

('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.

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

### **Example**

**import** React from 'react';

**import** ReactDOM from 'react-dom';

// Example Component

function Example()

{

**return**(<div>

            {

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

            }

           </div>

        );

}

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**

condition ?  **true** : **false**

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

### **Example**

render() {

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

**return** (

    <div>

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

    </div>

  );

}

## Switch case operator

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

### **Example**

function NotificationMsg({ text}) {

**switch**(text) {

**case** 'Hi All':

**return** <Message: text={text} />;

**case** 'Hello JavaTpoint':

**return** <Message text={text} />;

**default**:

**return** **null**;

  }

}

## 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**

function NotificationMsg({ text, state }) {

**return** (

    <div>

      {{

        info: <Message text={text} />,

        warning: <Message text={text} />,

      }[state]}

    </div>

  );

}

## 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**.

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

// Message Component

function Message(props)

{

**if** (props.isLoggedIn)

**return** <h1>Welcome Back!!!</h1>;

**else**

**return** <h1>Please Login First!!!</h1>;

}

// Login Component

function Login(props)

{

**return**(

           <button onClick = {props.clickInfo}> Login </button>

       );

}

// Logout Component

function Logout(props)

{

**return**(

           <button onClick = {props.clickInfo}> Logout </button>

       );

}

**class** App **extends** Component{

   constructor(props)

    {

**super**(props);

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

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

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

    }

   handleLogin()

    {

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

    }

    handleLogout()

    {

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

    }

    render(){

**return**(

            <div>

        <h1> Conditional Rendering Example </h1>

                <Message isLoggedIn = {**this**.state.isLoggedIn}/>

                {

                    (**this**.state.isLoggedIn)?(

                    <Logout clickInfo = {**this**.handleLogout} />

                    ) : (

                    <Login clickInfo = {**this**.handleLogin} />

                    )

                }

            </div>

            );

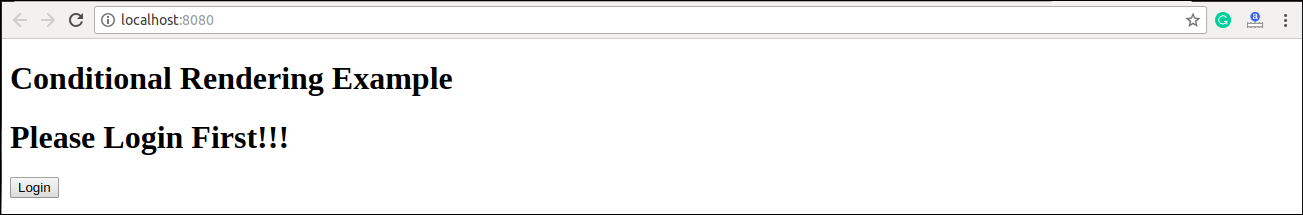
    }

}

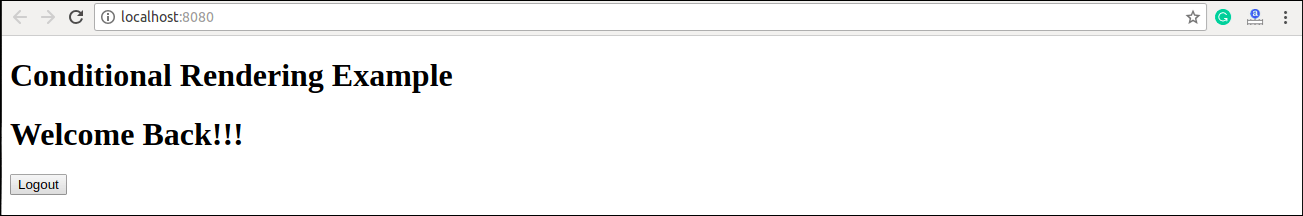
export **default** App;

**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.

**import** React from 'react';

**import** ReactDOM from 'react-dom';

function Show(props)

{

**if**(!props.displayMessage)

**return** **null**;

**else**

**return** <h3>Component is rendered</h3>;

}

ReactDOM.render(

    <div>

        <h1>Message</h1>

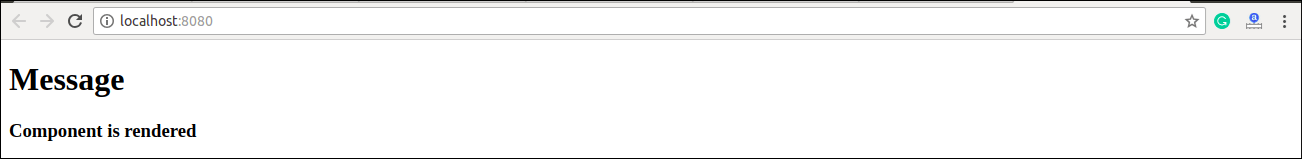
        <Show displayMessage = {**true**} />

    </div>,

    document.getElementById('app')

);

**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**

var numbers = [1, 2, 3, 4, 5];

**const** multiplyNums = numbers.map((number)=>{

**return** (number \* 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**

**import** React from 'react';

**import** ReactDOM from 'react-dom';

**const** myList = ['Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa'];

**const** listItems = myList.map((myList)=>{

**return** <li>{myList}</li>;

});

ReactDOM.render(

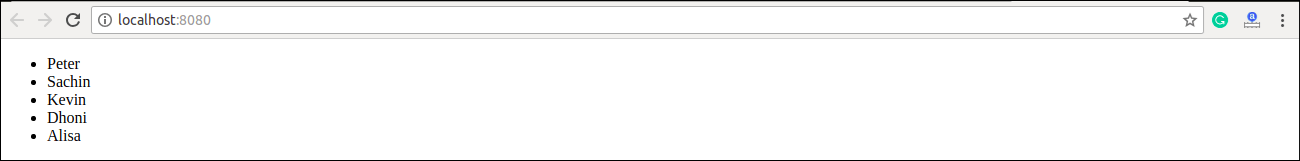
    <ul> {listItems} </ul>,

    document.getElementById('app')

);

export **default** App;

**Output**



## 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**

**import** React from 'react';

**import** ReactDOM from 'react-dom';

function NameList(props) {

**const** myLists = props.myLists;

**const** listItems = myLists.map((myList) =>

    <li>{myList}</li>

  );

**return** (

    <div>

        <h2>Rendering Lists inside component</h2>

              <ul>{listItems}</ul>

    </div>

  );

}

**const** myLists = ['Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa'];

ReactDOM.render(

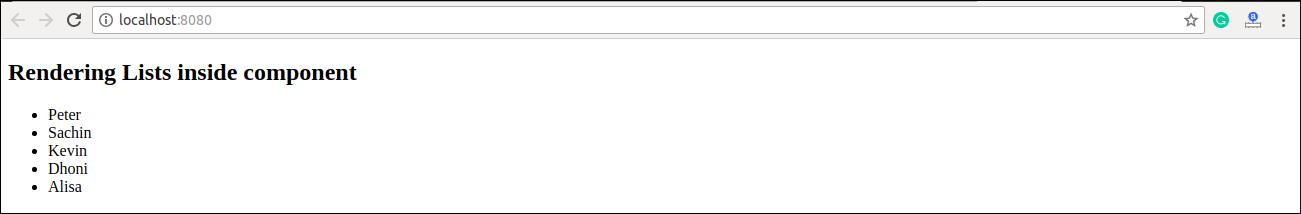
  <NameList myLists={myLists} />,

  document.getElementById('app')

);

export **default** App;

**Output**



# **React Keys**

A key is a unique identifier. In React, it is used to identify which items have changed, updated, or deleted from the Lists. It is useful when we dynamically created components or when the users alter the lists. It also helps to determine which components in a collection needs to be re-rendered instead of re-rendering the entire set of components every time.

Keys should be given inside the array to give the elements a stable identity. The best way to pick a key as a string that uniquely identifies the items in the list. It can be understood with the below example.

### **Example**

**const** stringLists = [ 'Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa' ];

**const** updatedLists = stringLists.map((strList)=>{

    <li key={strList.id}> {strList} </li>;

});

If there are no stable IDs for rendered items, you can assign the item **index** as a key to the lists. It can be shown in the below example.

### **Example**

**const** stringLists = [ 'Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa' ];

**const** updatedLists = stringLists.map((strList, index)=>{

    <li key={index}> {strList} </li>;

});

#### **Note: It is not recommended to use indexes for keys if the order of the item may change in future. It creates confusion for the developer and may cause issues with the component state.**

## Using Keys with component

Consider you have created a separate component for **ListItem** and extracting ListItem from that component. In this case, you should have to assign keys on the **<ListItem />** elements in the array, not to the **<li>** elements in the ListItem itself. To avoid mistakes, you have to keep in mind that keys only make sense in the context of the surrounding array. So, anything you are returning from map() function is recommended to be assigned a key.

### **Example: Incorrect Key usage**

**import** React from 'react';

**import** ReactDOM from 'react-dom';

function ListItem(props) {

**const** item = props.item;

**return** (

    // Wrong! No need to specify the key here.

    <li key={item.toString()}>

      {item}

    </li>

  );

}

function NameList(props) {

**const** myLists = props.myLists;

**const** listItems = myLists.map((strLists) =>

    // The key should have been specified here.

    <ListItem item={strLists} />

  );

**return** (

    <div>

        <h2>Incorrect Key Usage Example</h2>

              <ol>{listItems}</ol>

    </div>

  );

}

**const** myLists = ['Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa'];

ReactDOM.render(

  <NameList myLists={myLists}/>,

  document.getElementById('app')

);

export **default** App;

In the given example, the list is rendered successfully. But it is not a good practice that we had not assigned a key to the map() iterator.

**Output**



### **Example: Correct Key usage**

To correct the above example, we should have to assign key to the map() iterator.

**import** React from 'react';

**import** ReactDOM from 'react-dom';

function ListItem(props) {

**const** item = props.item;

**return** (

    // No need to specify the key here.

    <li> {item} </li>

  );

}

function NameList(props) {

**const** myLists = props.myLists;

**const** listItems = myLists.map((strLists) =>

   // The key should have been specified here.

    <ListItem key={myLists.toString()} item={strLists} />

  );

**return** (

    <div>

        <h2>Correct Key Usage Example</h2>

            <ol>{listItems}</ol>

    </div>

  );

}

**const** myLists = ['Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa'];

ReactDOM.render(

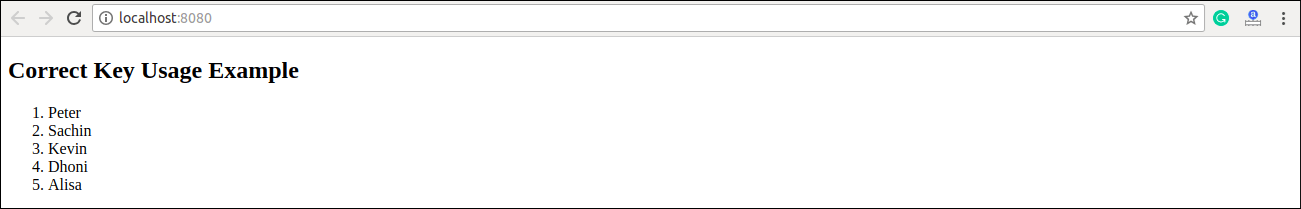
  <NameList myLists={myLists}/>,

  document.getElementById('app')

);

export **default** App;

**Output**



## Uniqueness of Keys among Siblings

We had discussed that keys assignment in arrays must be unique among their **siblings**. However, it doesn't mean that the keys should be **globally** unique. We can use the same set of keys in producing two different arrays. It can be understood in the below example.

### **Example**

**import** React from 'react';

**import** ReactDOM from 'react-dom';

function MenuBlog(props) {

**const** titlebar = (

    <ol>

      {props.data.map((show) =>

        <li key={show.id}>

          {show.title}

        </li>

      )}

    </ol>

  );

**const** content = props.data.map((show) =>

    <div key={show.id}>

      <h3>{show.title}: {show.content}</h3>

    </div>

  );

**return** (

    <div>

      {titlebar}

      <hr />

      {content}

    </div>

  );

}

**const** data = [

  {id: 1, title: 'First', content: 'Welcome to JavaTpoint!!'},

{id: 2, title: 'Second', content: 'It is the best ReactJS Tutorial!!'},

  {id: 3, title: 'Third', content: 'Here, you can learn all the ReactJS topics!!'}

];

ReactDOM.render(

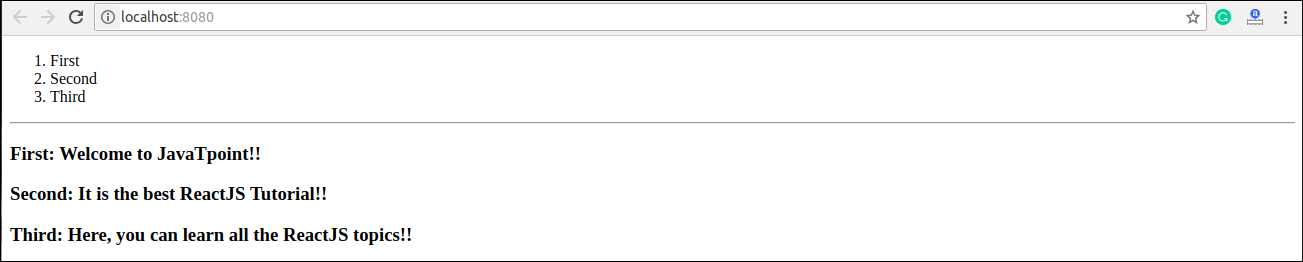
  <MenuBlog data={data} />,

  document.getElementById('app')

);

export **default** App;

**Output**



# **React Refs**

Refs is the shorthand used for **references** in React. It is similar to **keys** in React. It is an attribute which makes it possible to store a reference to particular DOM nodes or React elements. It provides a way to access React DOM nodes or React elements and how to interact with it. It is used when we want to change the value of a child component, without making the use of props.

## When to Use Refs

Refs can be used in the following cases:

* When we need DOM measurements such as managing focus, text selection, or media playback.
* It is used in triggering imperative animations.
* When integrating with third-party DOM libraries.
* It can also use as in callbacks.

## When to not use Refs

* Its use should be avoided for anything that can be done **declaratively**. For example, instead of using **open()** and **close()** methods on a Dialog component, you need to pass an **isOpen** prop to it.
* You should have to avoid overuse of the Refs.

## How to create Refs

In React, Refs can be created by using **React.createRef()**. It can be assigned to React elements via the **ref** attribute. It is commonly assigned to an instance property when a component is created, and then can be referenced throughout the component.

**class** MyComponent **extends** React.Component {

  constructor(props) {

**super**(props);

**this**.callRef = React.createRef();

  }

  render() {

**return** <div ref={**this**.callRef} />;

  }

}

## How to access Refs

In React, when a ref is passed to an element inside render method, a reference to the node can be accessed via the current attribute of the ref.

**const** node = **this**.callRef.current;

## Refs current Properties

The ref value differs depending on the type of the node:

* When the ref attribute is used in HTML element, the ref created with **React.createRef()** receives the underlying DOM element as its **current** property.
* If the ref attribute is used on a custom class component, then ref object receives the **mounted** instance of the component as its current property.
* The ref attribute cannot be used on **function components** because they don't have instances.

## Add Ref to DOM elements

In the below example, we are adding a ref to store the reference to a DOM node or element.

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

**import** { render } from 'react-dom';

**class** App **extends** React.Component {

  constructor(props) {

**super**(props);

**this**.callRef = React.createRef();

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

  }

  addingRefInput() {

**this**.callRef.current.focus();

  }

  render() {

**return** (

      <div>

        <h2>Adding Ref to DOM element</h2>

        <input

          type="text"

          ref={**this**.callRef} />

        <input

          type="button"

        value="Add text input"

          onClick={**this**.addingRefInput}

        />

      </div>

    );

  }

}

export **default** App;

**Output**



## Add Ref to Class components

In the below example, we are adding a ref to store the reference to a class component.

### **Example**

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

**import** { render } from 'react-dom';

function CustomInput(props) {

  let callRefInput = React.createRef();

  function handleClick() {

    callRefInput.current.focus();

  }

**return** (

    <div>

      <h2>Adding Ref to Class Component</h2>

      <input

        type="text"

        ref={callRefInput} />

      <input

        type="button"

        value="Focus input"

        onClick={handleClick}

      />

    </div>

  );

}

**class** App **extends** React.Component {

  constructor(props) {

**super**(props);

**this**.callRefInput = React.createRef();

  }

  focusRefInput() {

**this**.callRefInput.current.focus();

  }

  render() {

**return** (

      <CustomInput ref={**this**.callRefInput} />

    );

  }

}

export **default** App;

**Output**



## Callback refs

In react, there is another way to use refs that is called "**callback refs**" and it gives more control when the refs are **set** and **unset**. Instead of creating refs by createRef() method, React allows a way to create refs by passing a callback function to the ref attribute of a component. It looks like the below code.

<input type="text" ref={element => **this**.callRefInput = element} />

The callback function is used to store a reference to the DOM node in an instance property and can be accessed elsewhere. It can be accessed as below:

**this**.callRefInput.value

The example below helps to understand the working of callback refs.

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

**import** { render } from 'react-dom';

**class** App **extends** React.Component {

    constructor(props) {

**super**(props);

**this**.callRefInput = **null**;

**this**.setInputRef = element => {

**this**.callRefInput = element;

    };

**this**.focusRefInput = () => {

     //Focus the input using the raw DOM API

**if** (**this**.callRefInput) **this**.callRefInput.focus();

    };

  }

  componentDidMount() {

    //autofocus of the input on mount

**this**.focusRefInput();

  }

  render() {

**return** (

      <div>

    <h2>Callback Refs Example</h2>

        <input

          type="text"

          ref={**this**.setInputRef}  />

        <input

          type="button"

        value="Focus input text"

          onClick={**this**.focusRefInput} />

      </div>

    );

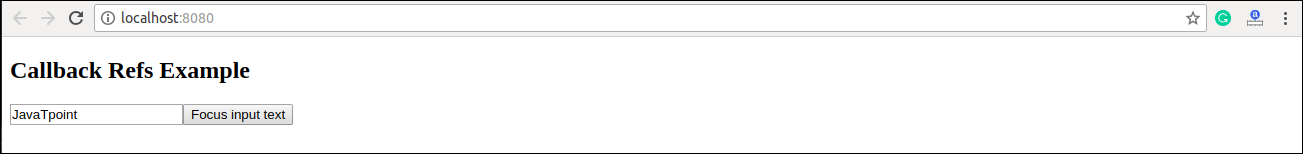
  }

}

export **default** App;

In the above example, React will call the "ref" callback to store the reference to the input DOM element when the component **mounts**, and when the component **unmounts**, call it with **null**. Refs are always **up-to-date** before the **componentDidMount** or **componentDidUpdate** fires. The callback refs pass between components is the same as you can work with object refs, which is created with React.createRef().

**Output**



## Forwarding Ref from one component to another component

Ref forwarding is a technique that is used for passing a **ref** through a component to one of its child components. It can be performed by making use of the **React.forwardRef()** method. This technique is particularly useful with **higher-order components** and specially used in reusable component libraries. The most common example is given below.

### **Example**

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

**import** { render } from 'react-dom';

**const** TextInput = React.forwardRef((props, ref) => (

  <input type="text" placeholder="Hello World" ref={ref} />

));

**const** inputRef = React.createRef();

**class** CustomTextInput **extends** React.Component {

 handleSubmit = e => {

    e.preventDefault();

    console.log(inputRef.current.value);

  };

  render() {

**return** (

      <div>

        <form onSubmit={e => **this**.handleSubmit(e)}>

          <TextInput ref={inputRef} />

          <button>Submit</button>

        </form>

      </div>

    );

  }

}

export **default** App;

In the above example, there is a component **TextInput** that has a child as an input field. Now, to pass or forward the **ref** down to the input, first, create a ref and then pass your ref down to **<TextInput ref={inputRef}>**. After that, React forwards the ref to the **forwardRef** function as a second argument. Next, we forward this ref argument down to **<input ref={ref}>**. Now, the value of the DOM node can be accessed at **inputRef.current**.

## React with useRef()

It is introduced in **React 16.7** and above version. It helps to get access the DOM node or element, and then we can interact with that DOM node or element such as focussing the input element or accessing the input element value. It returns the ref object whose **.current** property initialized to the passed argument. The returned object persist for the lifetime of the component.

### **Syntax**

**const** refContainer = useRef(initialValue);

### **Example**

In the below code, **useRef** is a function that gets assigned to a variable, **inputRef**, and then attached to an attribute called ref inside the HTML element in which you want to reference.

function useRefExample() {

**const** inputRef= useRef(**null**);

**const** onButtonClick = () => {

    inputRef.current.focus();

  };

**return** (

    <>

      <input ref={inputRef} type="text" />

      <button onClick={onButtonClick}>Submit</button>

    </>

  );

}

# **React Fragments**

In React, whenever you want to render something on the screen, you need to use a render method inside the component. This render method can return **single** elements or **multiple** elements. The render method will only render a single root node inside it at a time. However, if you want to return multiple elements, the render method will require a '**div**' tag and put the entire content or elements inside it. This extra node to the DOM sometimes results in the wrong formatting of your HTML output and also not loved by the many developers.

### **Example**

// Rendering with div tag

**class** App **extends** React.Component {

     render() {

**return** (

         //Extraneous div element

         <div>

           <h2> Hello World! </h2>

           <p> Welcome to the JavaTpoint. </p>

         </div>

      );

     }

}

To solve this problem, React introduced **Fragments** from the **16.2** and above version. Fragments allow you to group a list of children without adding extra nodes to the DOM.

### **Syntax**

<React.Fragment>

      <h2> child1 </h2>

    <p> child2 </p>

      .. ..... .... ...

</React.Fragment>

### **Example**

// Rendering with fragments tag

**class** App **extends** React.Component {

    render() {

**return** (

       <React.Fragment>

            <h2> Hello World! </h2>

        <p> Welcome to the JavaTpoint. </p>

         </React.Fragment>

     );

    }

}

## Why we use Fragments?

The main reason to use Fragments tag is:

1. It makes the execution of code faster as compared to the div tag.
2. It takes less memory.

## Fragments Short Syntax

There is also another shorthand exists for declaring fragments for the above method. It looks like **empty** tag in which we can use of '<>' and '' instead of the '**React.Fragment’.**

### **Example**

//Rendering with short syntax

**class** Columns **extends** React.Component {

  render() {

**return** (

      <>

        <h2> Hello World! </h2>

        <p> Welcome to the JavaTpoint </p>

      </>

    );

  }

}

## Keyed Fragments

The shorthand syntax does not accept key attributes. You need a key for mapping a collection to an array of fragments such as to create a description list. If you need to provide keys, you have to declare the fragments with the explicit <**React.Fragment**> syntax.

#### **Note: Key is the only attributes that can be passed with the Fragments.**

### **Example**

Function  = (props) {

**return** (

    <Fragment>

      {props.items.data.map(item => (

        // Without the 'key', React will give a key warning

        <React.Fragment key={item.id}>

          <h2>{item.name}</h2>

          <p>{item.url}</p>

          <p>{item.description}</p>

        </React.Fragment>

      ))}

    </Fragment>

  )

}

# **React Router**

Routing is a process in which a user is directed to different pages based on their action or request. ReactJS Router is mainly used for developing Single Page Web Applications. React Router is used to define multiple routes in the application. When a user types a specific URL into the browser, and if this URL path matches any 'route' inside the router file, the user will be redirected to that particular route.

React Router is a standard library system built on top of the React and used to create routing in the React application using React Router Package. It provides the synchronous URL on the browser with data that will be displayed on the web page. It maintains the standard structure and behavior of the application and mainly used for developing single page web applications.

## Need of React Router

React Router plays an important role to display multiple views in a single page application. Without React Router, it is not possible to display multiple views in React applications. Most of the social media websites like Facebook, Instagram uses React Router for rendering multiple views.

## React Router Installation

React contains three different packages for routing. These are:

1. **react-router:** It provides the core routing components and functions for the React Router applications.
2. **react-router-native:** It is used for mobile applications.
3. **react-router-dom:** It is used for web applications design.

It is not possible to install react-router directly in your application. To use react routing, first, you need to install react-router-dom modules in your application. The below command is used to install react router dom.

$ npm install react-router-dom --save

## Components in React Router

There are two types of router components:

* **<BrowserRouter>:** It is used for handling the dynamic URL.
* **<HashRouter>:** It is used for handling the static request.

### **Example**

**Step-1:** In our project, we will create two more components along with **App.js**, which is already present.

**About.js**

**import** React from 'react'

**class** About **extends** React.Component {

  render() {

**return** <h1>About</h1>

  }

}

export **default** About

**Contact.js**

**import** React from 'react'

**class** Contact **extends** React.Component {

  render() {

**return** <h1>Contact</h1>

  }

}

export **default** Contact

**App.js**

**import** React from 'react'

**class** App **extends** React.Component {

  render() {

**return** (

      <div>

        <h1>Home</h1>

      </div>

    )

  }

}

export **default** App

**Step-2:** For Routing, open the index.js file and import all the three component files in it. Here, you need to import line: **import { Route, Link, BrowserRouter as Router } from 'react-router-dom'** which helps us to implement the Routing. Now, our index.js file looks like below.

## What is Route?

It is used to define and render component based on the specified path. It will accept components and render to define what should be rendered.

**Index.js**

**import** React from 'react';

**import** ReactDOM from 'react-dom';

**import** { Route, Link, BrowserRouter as Router } from 'react-router-dom'

**import** './index.css';

**import** App from './App';

**import** About from './about'

**import** Contact from './contact'

**const** routing = (

  <Router>

    <div>

      <h1>React Router Example</h1>

      <Route path="/" component={App} />

      <Route path="/about" component={About} />

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

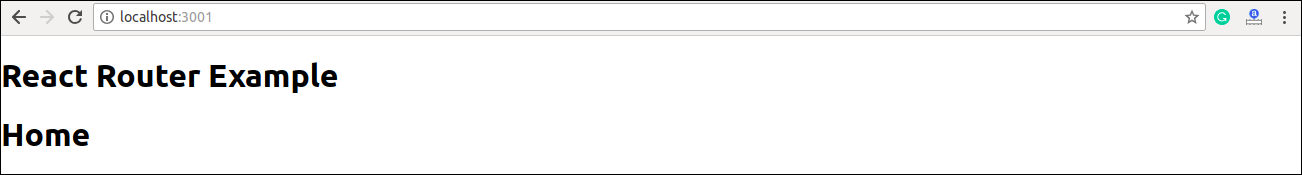
    </div>

  </Router>

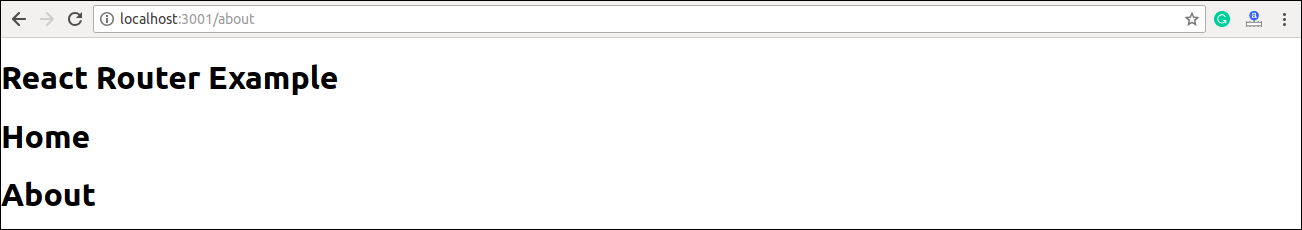
)

ReactDOM.render(routing, document.getElementById('root'));

**Step-3:** Open **command prompt**, go to your project location, and then type **npm start**. You will get the following screen.



Now, if you enter **manually** in the browser: **localhost:3000/about**, you will see **About** component is rendered on the screen.



**Step-4:** In the above screen, you can see that **Home** component is still rendered. It is because the home path is '**/**' and about path is '**/about**', so you can observe that **slash** is common in both paths which render both components. To stop this behavior, you need to use the **exact** prop. It can be seen in the below example.

**Index.js**

**import** React from 'react';

**import** ReactDOM from 'react-dom';

**import** { Route, Link, BrowserRouter as Router } from 'react-router-dom'

**import** './index.css';

**import** App from './App';

**import** About from './about'

**import** Contact from './contact'

**const** routing = (

  <Router>

    <div>

      <h1>React Router Example</h1>

      <Route exact path="/" component={App} />

      <Route path="/about" component={About} />

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

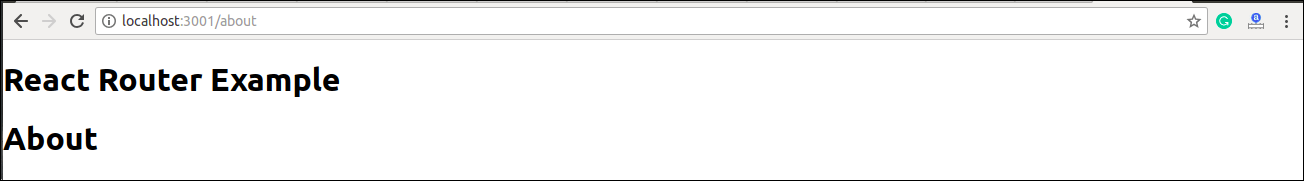
    </div>

  </Router>

)

ReactDOM.render(routing, document.getElementById('root'));

**Output**



## Adding Navigation using Link component

Sometimes, we want to need **multiple** links on a single page. When we click on any of that particular **Link**, it should load that page which is associated with that path without **reloading** the web page. To do this, we need to import **<Link>** component in the **index.js** file.

### **What is < Link> component?**

This component is used to create links which allow to **navigate** on different **URLs** and render its content without reloading the webpage.

**Example**

**Index.js**

**import** React from 'react';

**import** ReactDOM from 'react-dom';

**import** { Route, Link, BrowserRouter as Router } from 'react-router-dom'

**import** './index.css';

**import** App from './App';

**import** About from './about'

**import** Contact from './contact'

**const** routing = (

  <Router>

    <div>

      <h1>React Router Example</h1>

      <ul>

        <li>

          <Link to="/">Home</Link>

        </li>

        <li>

          <Link to="/about">About</Link>

       </li>

        <li>

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

        </li>

      </ul>

      <Route exact path="/" component={App} />

      <Route path="/about" component={About} />

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

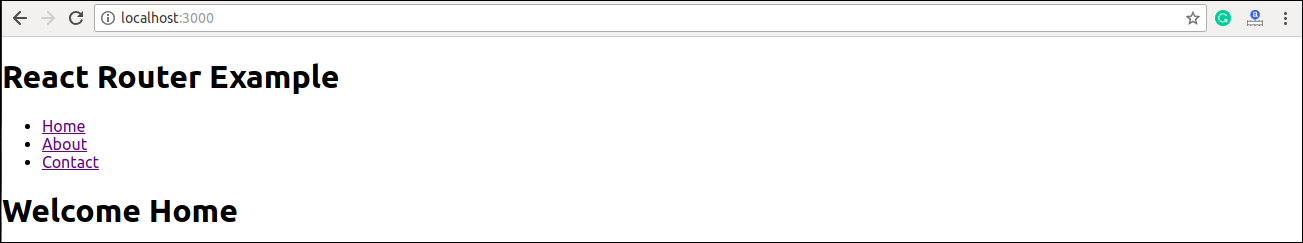
    </div>

  </Router>

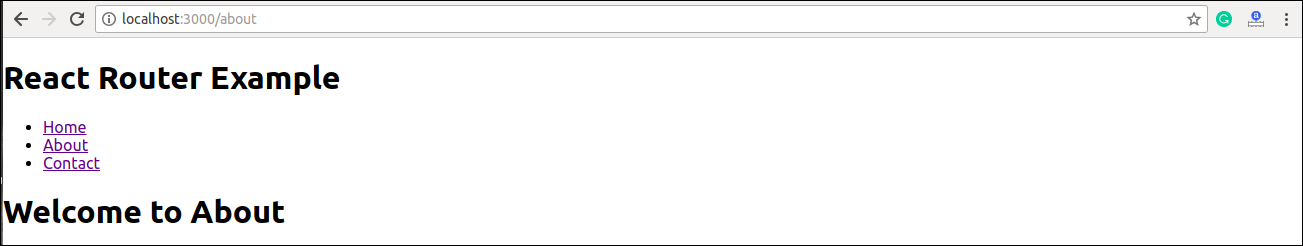
)

ReactDOM.render(routing, document.getElementById('root'));

**Output**



After adding Link, you can see that the routes are rendered on the screen. Now, if you click on the **About**, you will see URL is changing and About component is rendered.



Now, we need to add some **styles** to the Link. So that when we click on any particular link, it can be easily **identified** which Link is **active**. To do this react router provides a new trick **NavLink** instead of **Link**. Now, in the index.js file, replace Link from Navlink and add properties **activeStyle**. The activeStyle properties mean when we click on the Link, it should have a specific style so that we can differentiate which one is currently active.

**import** React from 'react';

**import** ReactDOM from 'react-dom';

**import** { BrowserRouter as Router, Route, Link, NavLink } from 'react-router-dom'

**import** './index.css';

**import** App from './App';

**import** About from './about'

**import** Contact from './contact'

**const** routing = (

  <Router>

    <div>

      <h1>React Router Example</h1>

      <ul>

        <li>

          <NavLink to="/" exact activeStyle={

             {color:'red'}

          }>Home</NavLink>

        </li>

        <li>

          <NavLink to="/about" exact activeStyle={

             {color:'green'}

          }>About</NavLink>

        </li>

        <li>

          <NavLink to="/contact" exact activeStyle={

             {color:'magenta'}

          }>Contact</NavLink>

        </li>

      </ul>

      <Route exact path="/" component={App} />

      <Route path="/about" component={About} />

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

    </div>

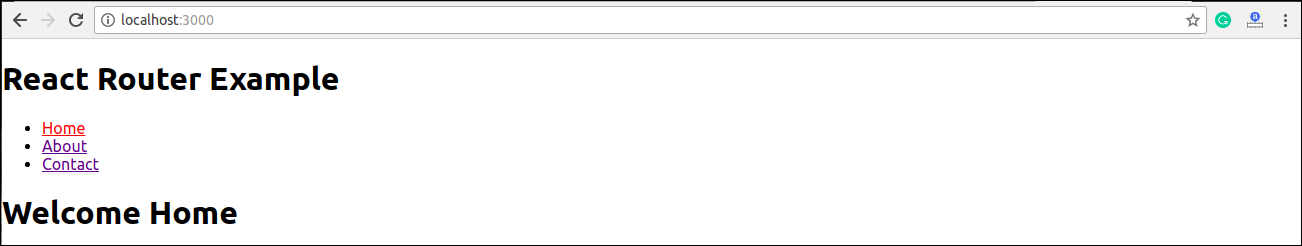
  </Router>

)

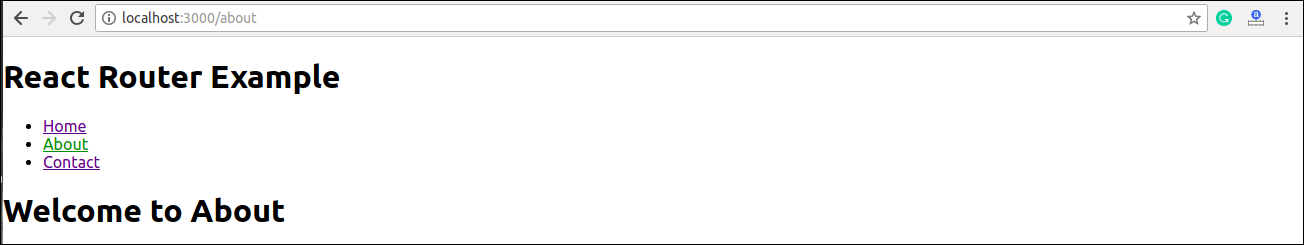
ReactDOM.render(routing, document.getElementById('root'));

**Output**

When we execute the above program, we will get the following screen in which we can see that **Home** link is of color **Red** and is the only currently **active** link.



Now, when we click on **About** link, its color shown **green** that is the currently **active** link.



### **<Link> vs <NavLink>**

The Link component allows navigating the different routes on the websites, whereas NavLink component is used to add styles to the active routes.

## React Router Switch

The <**Switch**> component is used to render components only when the path will be **matched**. Otherwise, it returns to the **not found** component.

To understand this, first, we need to create a **notfound** component.

**notfound.js**

**import** React from 'react'

**const** Notfound = () => <h1>Not found</h1>

export **default** Notfound

Now, import component in the index.js file. It can be seen in the below code.

**Index.js**

**import** React from 'react';

**import** ReactDOM from 'react-dom';

**import** { BrowserRouter as Router, Route, Link, NavLink, Switch } from 'react-router-dom'

**import** './index.css';

**import** App from './App';

**import** About from './about'

**import** Contact from './contact'

**import** Notfound from './notfound'

**const** routing = (

  <Router>

    <div>

      <h1>React Router Example</h1>

      <ul>

        <li>

          <NavLink to="/" exact activeStyle={

             {color:'red'}

          }>Home</NavLink>

        </li>

        <li>

          <NavLink to="/about" exact activeStyle={

             {color:'green'}

          }>About</NavLink>

        </li>

        <li>

          <NavLink to="/contact" exact activeStyle={

             {color:'magenta'}

          }>Contact</NavLink>

        </li>

      </ul>

      <Switch>

         <Route exact path="/" component={App} />

         <Route path="/about" component={About} />

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

         <Route component={Notfound} />

      </Switch>

    </div>

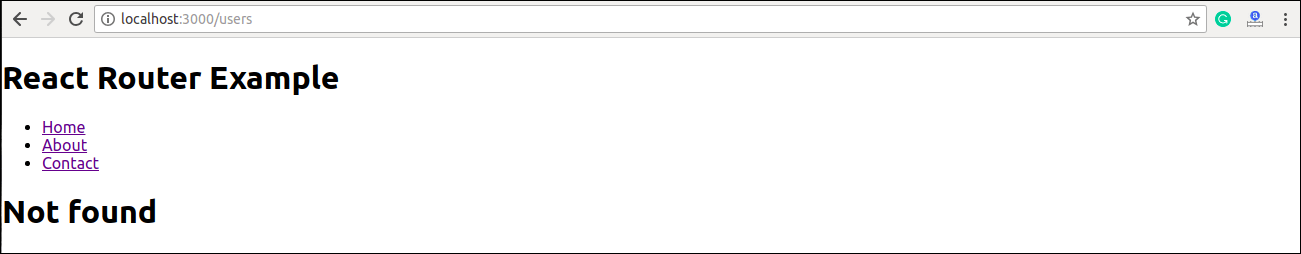
  </Router>

)

ReactDOM.render(routing, document.getElementById('root'));

**Output**

If we manually enter the **wrong** path, it will give the not found error.



## React Router <Redirect>

A <Redirect> component is used to redirect to another route in our application to maintain the old URLs. It can be placed anywhere in the route hierarchy.

### **Nested Routing in React**

Nested routing allows you to render **sub-routes** in your application. It can be understood in the below example.

**Example**

**index.js**

**import** React from 'react';

**import** ReactDOM from 'react-dom';

**import** { BrowserRouter as Router, Route, Link, NavLink, Switch } from 'reactrouter-dom'

**import** './index.css';

**import** App from './App';

**import** About from './about'

**import** Contact from './contact'

**import** Notfound from './notfound'

**const** routing = (

  <Router>

    <div>

      <h1>React Router Example</h1>

      <ul>

        <li>

          <NavLink to="/" exact activeStyle={

             {color:'red'}

          }>Home</NavLink>

        </li>

        <li>

          <NavLink to="/about" exact activeStyle={

             {color:'green'}

          }>About</NavLink>

        </li>

        <li>

          <NavLink to="/contact" exact activeStyle={

             {color:'magenta'}

          }>Contact</NavLink>

     </li>

      </ul>

      <Switch>

         <Route exact path="/" component={App} />

         <Route path="/about" component={About} />

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

         <Route component={Notfound} />

      </Switch>

    </div>

  </Router>

)

ReactDOM.render(routing, document.getElementById('root'));

In the **contact.js** file, we need to import the **React Router** component to implement the **subroutes**.

**contact.js**

**import** React from 'react'

**import** { Route, Link } from 'react-router-dom'

**const** Contacts = ({ match }) => <p>{match.params.id}</p>

**class** Contact **extends** React.Component {

  render() {

**const** { url } = **this**.props.match

**return** (

      <div>

        <h1>Welcome to Contact Page</h1>

        <strong>Select contact Id</strong>

        <ul>

          <li>

            <Link to="/contact/1">Contacts 1 </Link>

          </li>

          <li>

            <Link to="/contact/2">Contacts 2 </Link>

          </li>

          <li>

            <Link to="/contact/3">Contacts 3 </Link>

          </li>

          <li>

            <Link to="/contact/4">Contacts 4 </Link>

          </li>

        </ul>

        <Route path="/contact/:id" component={Contacts} />

      </div>

    )

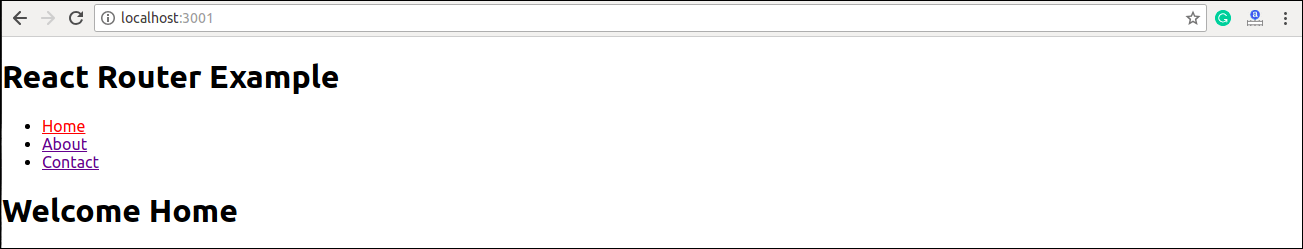
  }

}

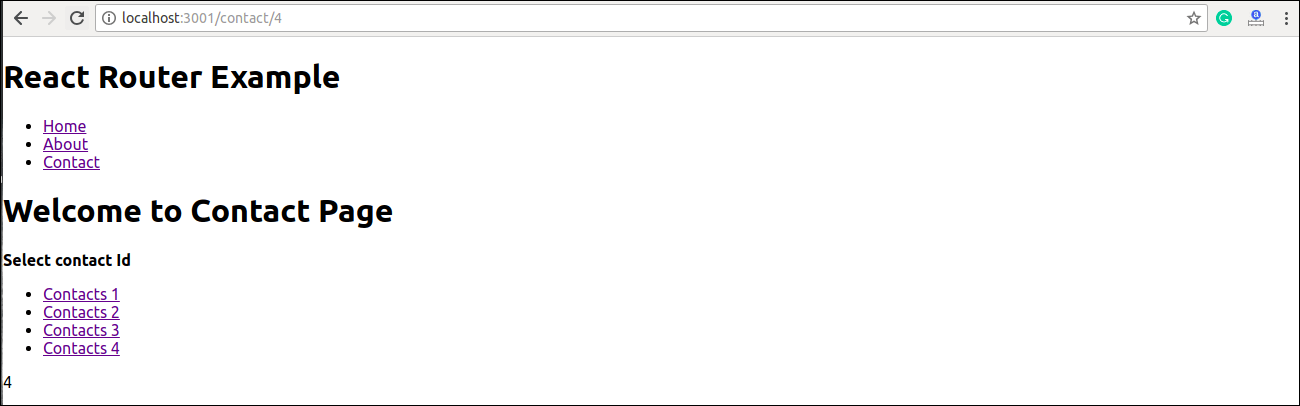
export **default** Contact

**Output**

When we execute the above program, we will get the following output.



After clicking the **Contact** link, we will get the contact list. Now, selecting any contact, we will get the corresponding output. It can be shown in the below example.



## Benefits Of React Router

The benefits of React Router is given below:

* In this, it is not necessary to set the browser history manually.
* Link uses to navigate the internal links in the application. It is similar to the anchor tag.
* It uses Switch feature for rendering.
* The Router needs only a Single Child element.
* In this, every component is specified in .

# **React CSS**

CSS in React is used to style the React App or Component. The **style** attribute is the most used attribute for styling in React applications, which adds dynamically-computed styles at render time. It accepts a JavaScript object in **camelCased** properties rather than a CSS string. There are many ways available to add styling to your React App or Component with CSS. Here, we are going to discuss mainly **four** ways to style React Components, which are given below:

1. Inline Styling
2. CSS Stylesheet
3. CSS Module
4. Styled Components

## 1. Inline Styling

The inline styles are specified with a JavaScript object in camelCase version of the style name. Its value is the style?s value, which we usually take in a string.

### **Example**

**App.js**

**import** React from 'react';

**import** ReactDOM from 'react-dom';

**class** App **extends** React.Component {

  render() {

**return** (

      <div>

      <h1 style={{color: "Green"}}>Hello JavaTpoint!</h1>

      <p>Here, you can find all CS tutorials.</p>

      </div>

    );

  }

}

export **default** App;

#### Note:**You can see in the above example, we have used two curly braces in:** <h1 style={{color: "Green"}}>Hello JavaTpoint!</h1>**. It is because, in JSX, JavaScript expressions are written inside curly braces, and JavaScript objects also use curly braces, so the above styling is written inside two sets of curly braces {{}}.**

**Output**



### **camelCase Property Name**

If the properties have two names, like **background-color**, it must be written in camel case syntax.

**Example**

**App.js**

**import** React from 'react';

**import** ReactDOM from 'react-dom';

**class** App **extends** React.Component {

  render() {

**return** (

      <div>

      <h1 style={{color: "Red"}}>Hello JavaTpoint!</h1>

<p style={{backgroundColor: "lightgreen"}}>Here, you can find all CS tutorials.</p>

      </div>

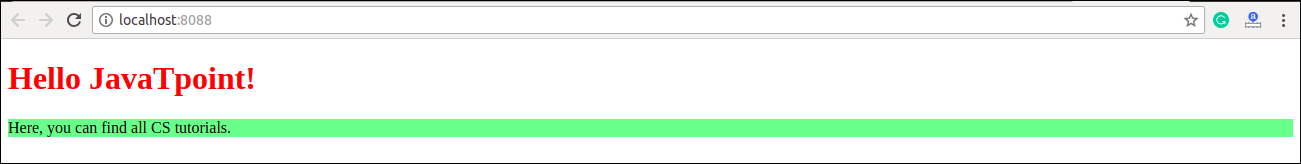
    );

  }

}

export **default** App;

**Output**



### **Using JavaScript Object**

The inline styling also allows us to create an object with styling information and refer it in the style attribute.

**Example**

**App.js**

**import** React from 'react';

**import** ReactDOM from 'react-dom';

**class** App **extends** React.Component {

  render() {

**const** mystyle = {

       color: "Green",

      backgroundColor: "lightBlue",

      padding: "10px",

      fontFamily: "Arial"

    };

**return** (

      <div>

      <h1 style={mystyle}>Hello JavaTpoint</h1>

      <p>Here, you can find all CS tutorials.</p>

      </div>

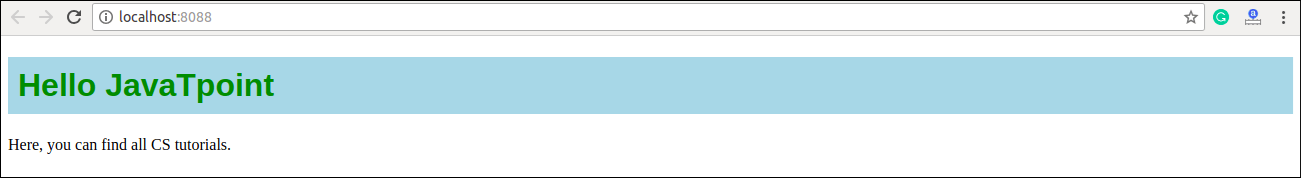
    );

  }

}

export **default** App;

**Output**



## 2. CSS Stylesheet

You can write styling in a separate file for your React application, and save the file with a .css extension. Now, you can **import** this file in your application.

### **Example**

**App.js**

**import** React from 'react';

**import** ReactDOM from 'react-dom';

**import** './App.css';

**class** App **extends** React.Component {

  render() {

**return** (

      <div>

      <h1>Hello JavaTpoint</h1>

      <p>Here, you can find all CS tutorials.</p>

      </div>

    );

  }

}

export **default** App;

**App.css**

body {

  background-color: #008080;

  color: yellow;

  padding: 40px;

  font-family: Arial;

  text-align: center;

}

**Index.html**

1. <!DOCTYPE html>

<html lang="en">

  <head>

    <meta charset="utf-8" />

    <meta name="viewport"

      content="width=device-width, initial-scale=1" />

    <title>React App</title>

  </head>

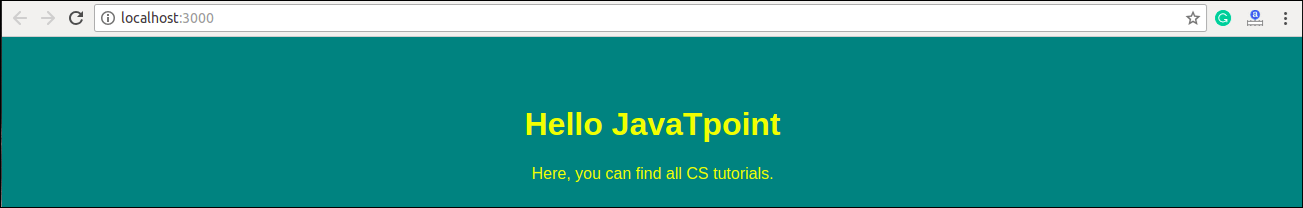
  <body>

    <div id="app"></div>

  </body>

</html>

**Output**



## 3. CSS Module

CSS Module is another way of adding styles to your application. It is a **CSS file** where all class names and **animation** names are scoped locally by default. It is available only for the component which imports it, means any styling you add can never be applied to other components without your permission, and you never need to worry about name conflicts. You can create CSS Module with the **.module.css** extension like a **myStyles.module.css** name.

### **Example**

**App.js**

**import** React from 'react';

**import** ReactDOM from 'react-dom';

**import** styles from './myStyles.module.css';

**class** App **extends** React.Component {

  render() {

**return** (

      <div>

      <h1 className={styles.mystyle}>Hello JavaTpoint</h1>

      <p className={styles.parastyle}>It provides great CS tutorials.</p>

      </div>

    );

  }

}

export **default** App;

**myStyles.module.css**

.mystyle {

  background-color: #cdc0b0;

  color: Red;

  padding: 10px;

  font-family: Arial;

  text-align: center;

}

.parastyle{

  color: Green;

  font-family: Arial;

  font-size: 35px;

  text-align: center;

}

**Output**



## 4. Styled Components

Styled-components is a **library** for React. It uses enhance CSS for styling React component systems in your application, which is written with a mixture of JavaScript and CSS.

**The styled-components provides:**

* Automatic critical CSS
* No class name bugs
* Easier deletion of CSS
* Simple dynamic styling
* Painless maintenance

### **Installation**

The styled-components library takes a single command to install in your React application. which is:

1. $ npm install styled-components --save

**Example**

Here, we create a variable by selecting a particular HTML element such as **<div>**, **<Title>**, and **<paragraph>** where we store our style attributes. Now we can use the name of our variable as a wrapper **<Div></Div>** kind of React component.

**App.js**

**import** React from 'react';

**import** ReactDOM from 'react-dom';

**import** styled from 'styled-components';

**class** App **extends** React.Component {

  render() {

**const** Div:any = styled.div`

            margin: 20px;

            border: 5px dashed green;

            &:hover {

            background-color: ${(props:any) => props.hoverColor};

            }

            `;

**const** Title = styled.h1`

            font-family: Arial;

            font-size: 35px;

            text-align: center;

         color: palevioletred;

            `;

**const** Paragraph = styled.p`

            font-size: 25px;

            text-align: center;

            background-Color: lightgreen;

            `;

**return** (

       <div>

            <Title>Styled Components Example</Title>

           <p></p>

            <Div hoverColor="Orange">

                 <Paragraph>Hello JavaTpoint!!</Paragraph>

            </Div>

        </div>

    );

  }

}

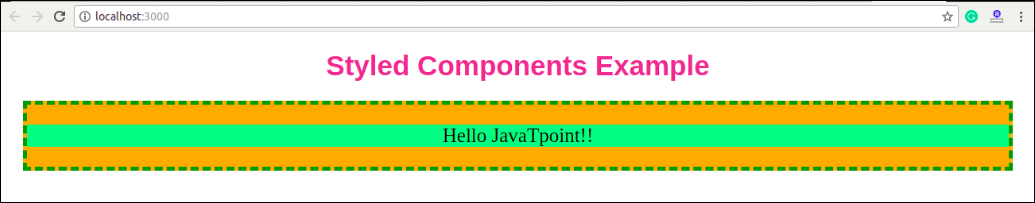
export **default** App;

**Output**

Now, execute the App.js file, we will get the output as shown below.



When we move the mouse pointer over the image, its color will be changed, as shown in the below image.



# **React Animation**

The animation is a technique in which images are manipulated to appear as moving images. It is one of the most used technique to make an interactive web application. In React, we can add animation using an explicit group of components known as the **React Transition Group**.

React Transition Group is an add-on component for managing component states and useful for defining **entering** and **exiting** transitions. It is not able to animate styles by itself. Instead, it exposes transition states, manages classes and group elements, and manipulates the DOM in useful ways. It makes the implementation of visual transitions much easier.

React Transition group has mainly **two APIs** to create transitions. These are:

1. **ReactTransitionGroup:** It uses as a low-level API for animation.
2. **ReactCSSTransitionGroup:** It uses as a high-level API for implementing basic CSS transitions and animations.

## Installation

We need to install **react-transition-group** for creating animation in React Web application. You can use the below command.

1. $ npm install react-transition-group --save

## React Transition Group Components

React Transition Group API provides **three** main components. These are:

1. Transition
2. CSSTransition
3. Transition Group

### **Transition**

It has a simple component API to describe a transition from one component state to another over time. It is mainly used to animate the **mounting** and **unmounting** of a component. It can also be used for in-place transition states as well.

We can access the Transition component into four states:

* entering
* entered
* exiting
* exited

### **CSSTransition**

The CSSTransition component uses CSS stylesheet classes to write the transition and create animations. It is inspired by the **ng-animate** library. It can also inherit all the props of the transition component. We can divide the "CSSTransition" into **three** states. These are:

* Appear
* Enter
* Exit

CSSTransition component must be applied in a pair of class names to the child components. The first class is in the form of **name-stage** and the second class is in the **name-stage-active**. For example, you provide the name fade, and when it applies to the 'enter' stage, the two classes will be **fade-enter** and **fade-enter-active**. It may also take a prop as Timeout which defines the maximum time to animate.

### **TransitionGroup**

This component is used to manage a set of transition components (Transition and CSSTransition) in a list. It is a state machine that controls the **mounting** and **unmounting** of components over time. The Transition component does not define any animation directly. Here, how 'list' item animates is based on the individual transition component. It means, the "TransitionGroup" component can have different animation within a component.

Let us see the example below, which clearly help to understand the React Animation.

**Example**

**App.js**

In the App.js file, import react-transition-group component, and create the CSSTransition component that uses as a wrapper of the component you want to animate. We are going to use **transitionEnterTimeout** and **transitionLeaveTimeout** for CSS Transition. The Enter and Leave animations used when we want to insert or delete elements from the list.

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

**import** { CSSTransitionGroup } from 'react-transition-group';

**class** App **extends** React.Component {

    constructor(props) {

**super**(props);

**this**.state = {items: ['Blockchain', 'ReactJS', 'TypeScript', 'JavaTpoint']};

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

  }

  handleAdd() {

**const** newItems = **this**.state.items.concat([

      prompt('Enter Item Name')

    ]);

**this**.setState({items: newItems});

  }

  handleRemove(i) {

    let newItems = **this**.state.items.slice();

    newItems.splice(i, 1);

**this**.setState({items: newItems});

  }

  render() {

**const** items = **this**.state.items.map((item, i) => (

      <div key={item} onClick={() => **this**.handleRemove(i)}>

        {item}

      </div>

    ));

**return** (

      <div>

    <h1>Animation Example</h1>

            <button onClick={**this**.handleAdd}>Insert Item</button>

            <CSSTransitionGroup

               transitionName="example"

           transitionEnterTimeout={800}

               transitionLeaveTimeout={600}>

               {items}

            </CSSTransitionGroup>

      </div>

    );

  }

}

export **default** App;

**Main.js**

**import** React from 'react';

**import** ReactDOM from 'react-dom';

**import** App from './App.js';

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

**style.css**

Add style.css file in your application, and add the following CSS styles. Now, to use this CSS file, you need to add the **link** of this file in your HTML file.

.example-enter {

  opacity: 0.01;

}

.example-enter.example-enter-active {

  opacity: 1;

  transition: opacity 500ms ease-in;

}

.example-leave {

  opacity: 1;

}

.example-leave.example-leave-active {

  opacity: 0.01;

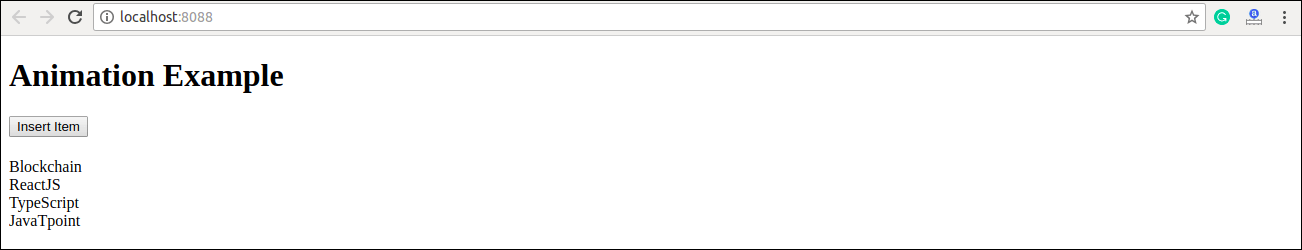
  transition: opacity 300ms ease-in;

}

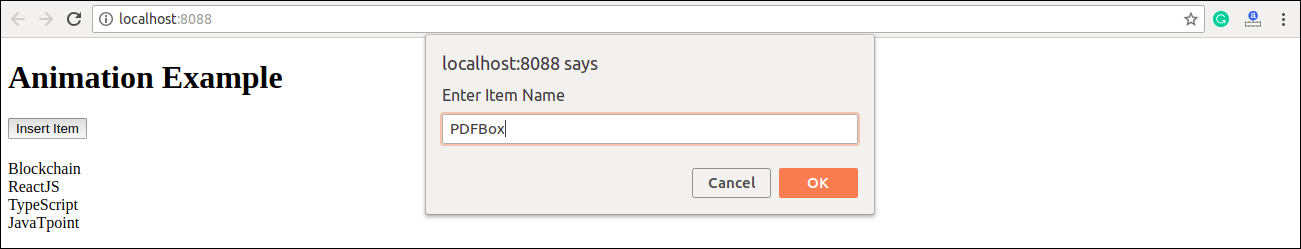
In the above example, the animation durations are specified in both the **CSS** and **render** method. It tells React component when to remove the animation classes from the list and if it is leaving when to remove the element from the DOM.

**Output**

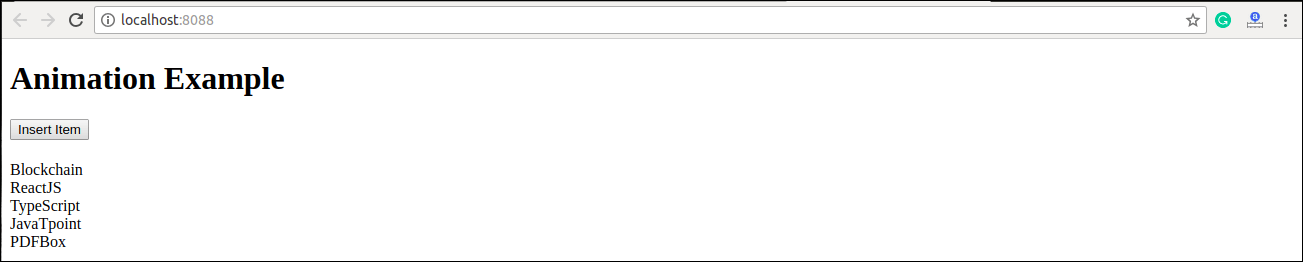
When we execute the above program, it gives the below output.



Click on '**Insert Item**' button, the following screen appears.



Once we insert the item and press **Ok**, the new item can be added in the list with fade in style. Here, we can also delete any item from the list by clicking on the particular link.



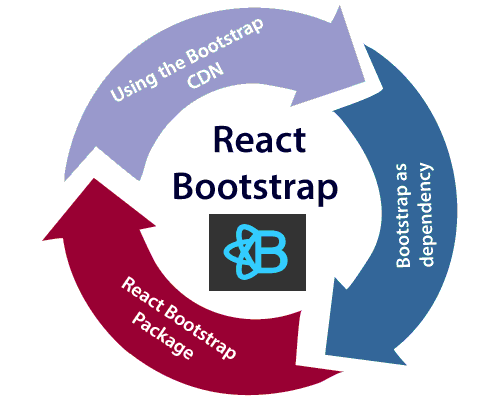
# **React Bootstrap**

Single-page applications gaining popularity over the last few years, so many front-end frameworks have introduced such as Angular, React, Vue.js, Ember, etc. As a result, jQuery is not a necessary requirement for building web apps. Today, React has the most used JavaScript framework for building web applications, and Bootstrap become the most popular CSS framework. So, it is necessary to learn various ways in which Bootstrap can be used in React apps, which is the main aim of this section.

## Adding Bootstrap for React

We can add Bootstrap to the React app in several ways. The **three** most common ways are given below:

1. Using the Bootstrap CDN
2. Bootstrap as Dependency
3. React Bootstrap Package



## Using the Bootstrap CDN

It is the easiest way of adding Bootstrap to the React app. There is no need to install or download Bootstrap. We can simply put an **<link>** into the **<head>** section of the **index.html** file of the React app as shown in the following snippet.

<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css" integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T" crossorigin="anonymous">

If there is a need to use Bootstrap components which depend on JavaScript/jQuery in the React application, we need to include **jQuery**, **Popper.js**, and **Bootstrap.js** in the document. Add the following imports in the **<script>** tags near the end of the closing **</body>** tag of the **index.html** file.

<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="sha384-q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo" crossorigin="anonymous"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js" integrity="sha384-UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIHNDz0W1" crossorigin="anonymous"></script>

<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js" integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM" crossorigin="anonymous"></script>

In the above snippet, we have used jQuery's slim version, although we can also use the full version as well. Now, Bootstrap is successfully added in the React application, and we can use all the CSS utilities and UI components available from Bootstrap in the React application.

## Bootstrap as Dependency

If we are using a build tool or a module bundler such as Webpack, then importing Bootstrap as dependency is the preferred option for adding Bootstrap to the React application. We can install Bootstrap as a dependency for the React app. To install the Bootstrap, run the following commands in the terminal window.

1. $ npm install bootstrap --save

Once Bootstrap is installed, we can import it in the React application entry file. If the React project created using the **create-react-app** tool, open the **src/index.js** file, and add the following code:

1. **import** 'bootstrap/dist/css/bootstrap.min.css';

Now, we can use the CSS classes and utilities in the React application. Also, if we want to use the JavaScript components, we need to install the **jquery** and **popper.js** packages from **npm**. To install the following packages, run the following command in the terminal window.

1. $ npm install jquery popper.js

Next, go to the **src/index.js** file and add the following imports.

1. **import** $ from 'jquery';
2. **import** Popper from 'popper.js';
3. **import** 'bootstrap/dist/js/bootstrap.bundle.min';

Now, we can use Bootstrap JavaScript Components in the React application.

## React Bootstrap Package

The React Bootstrap package is the most popular way to add bootstrap in the React application. There are many Bootstrap packages built by the community, which aim to rebuild Bootstrap components as React components. The **two** most popular Bootstrap packages are:

1. **react-bootstrap:** It is a complete re-implementation of the Bootstrap components as React components. It does not need any dependencies like bootstrap.js or jQuery. If the React setup and React-Bootstrap installed, we have everything which we need.
2. **reactstrap:** It is a library which contains React Bootstrap 4 components that favor composition and control. It does not depend on jQuery or Bootstrap JavaScript. However, react-popper is needed for advanced positioning of content such as Tooltips, Popovers, and auto-flipping Dropdowns.

## React Bootstrap Installation

Let us create a new React app using the **create-react-app** command as follows.

1. $ npx create-react-app react-bootstrap-app

After creating the React app, the best way to install Bootstrap is via the npm package. To install Bootstrap, navigate to the React app folder, and run the following command.

1. $ npm install react-bootstrap bootstrap --save

### **Importing Bootstrap**

Now, open the **src/index.js** file and add the following code to import the Bootstrap file.

1. **import** 'bootstrap/dist/css/bootstrap.min.css';

We can also import individual components **like import { SplitButton, Dropdown } from 'react-bootstrap';** instead of the entire library. It provides the specific components which we need to use, and can significantly reduce the amount of code.

In the React app, create a new file named **ThemeSwitcher.js** in the **src** directory and put the following code.

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

**import** { SplitButton, Dropdown } from 'react-bootstrap';

**class** ThemeSwitcher **extends** Component {

  state = { theme: **null** }

  chooseTheme = (theme, evt) => {

    evt.preventDefault();

**if** (theme.toLowerCase() === 'reset') { theme = **null** }

**this**.setState({ theme });

  }

  render() {

**const** { theme } = **this**.state;

**const** themeClass = theme ? theme.toLowerCase() : 'default';

**const** parentContainerStyles = {

      position: 'absolute',

      height: '100%',

      width: '100%',

      display: 'table'

    };

**const** subContainerStyles = {

      position: 'relative',

      height: '100%',

      width: '100%',

      display: 'table-cell',

    };

**return** (

      <div style={parentContainerStyles}>

        <div style={subContainerStyles}>

          <span className={`h1 center-block text-center text-${theme ? themeClass : 'muted'}`} style={{ marginBottom: 25 }}>{theme || 'Default'}</span>

          <div className="center-block text-center">

            <SplitButton bsSize="large" bsStyle={themeClass} title={`${theme || 'Default Block'} Theme`}>

              <Dropdown.Item eventKey="Primary Block" onSelect={**this**.chooseTheme}>Primary Theme</Dropdown.Item>

              <Dropdown.Item eventKey="Danger Block" onSelect={**this**.chooseTheme}>Danger Theme</Dropdown.Item>

              <Dropdown.Item eventKey="Success Block" onSelect={**this**.chooseTheme}>Success Theme</Dropdown.Item>

              <Dropdown.Item divider />

              <Dropdown.Item eventKey="Reset Block" onSelect={**this**.chooseTheme}>Default Theme</Dropdown.Item>

            </SplitButton>

          </div>

        </div>

      </div>

    );

  }

}

export **default** ThemeSwitcher;

Now, update the **src/index.js** file with the following snippet.

**Index.js**

**import** 'bootstrap/dist/css/bootstrap.min.css';

**import** React from 'react';

**import** ReactDOM from 'react-dom';

**import** App from './App.js';

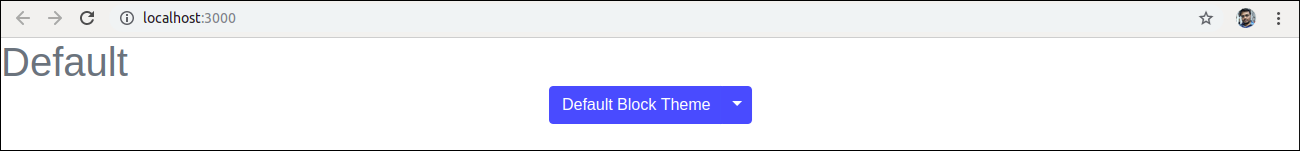
**import** './index.css';

**import** ThemeSwitcher from './ThemeSwitcher';

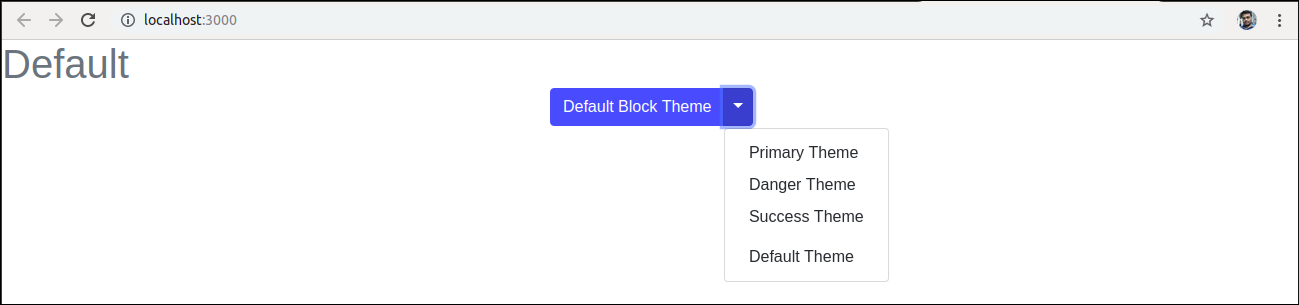
ReactDOM.render(<ThemeSwitcher />, document.getElementById('root'));

**Output**

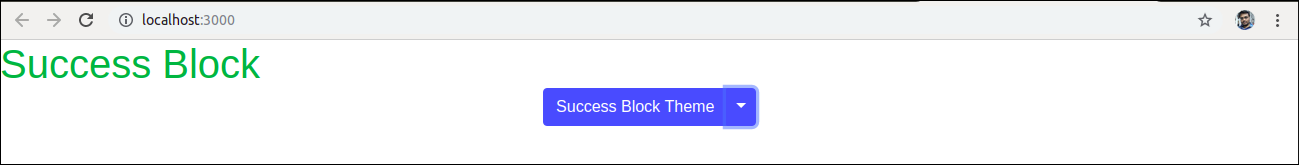
When we execute the React app, we should get the output as below.



Click on the dropdown menu. We will get the following screen.



Now, if we choose the **Success Theme**, we will get the below screen.



## Using reactstrap

Let us create a new React app using the create-react-app command as follows.

1. $ npx create-react-app reactstrap-app

Next, install the **reactstrap** via the npm package. To install reactstrap, navigate to the React app folder, and run the following command.

1. $ npm install bootstrap reactstrap --save

### **Importing Bootstrap**

Now, open the **src/index.js** file and add the following code to import the Bootstrap file.

1. **import** 'bootstrap/dist/css/bootstrap.min.css';

We can also import individual components **like import { Button, Dropdown } from 'reactstrap';** instead of the entire library. It provides the specific components which we need to use, and can significantly reduce the amount of code.

In the React app, create a new file named **ThemeSwitcher.js** in the **src** directory and put the following code.

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

**import** { Button, ButtonDropdown, DropdownToggle, DropdownMenu, DropdownItem } from 'reactstrap';

**class** ThemeSwitcher **extends** Component {

  state = { theme: **null**, dropdownOpen: **false** }

  toggleDropdown = () => {

**this**.setState({ dropdownOpen: !**this**.state.dropdownOpen });

  }

  resetTheme = evt => {

   evt.preventDefault();

**this**.setState({ theme: **null** });

  }

  chooseTheme = (theme, evt) => {

    evt.preventDefault();

**this**.setState({ theme });

  }

  render() {

**const** { theme, dropdownOpen } = **this**.state;

**const** themeClass = theme ? theme.toLowerCase() : 'secondary';

**return** (

      <div className="d-flex flex-wrap justify-content-center align-items-center">

        <span className={`h1 mb-4 w-100 text-center text-${themeClass}`}>{theme || 'Default'}</span>

        <ButtonDropdown isOpen={dropdownOpen} toggle={**this**.toggleDropdown}>

          <Button id="caret" color={themeClass}>{theme || 'Custom'} Theme</Button>

         <DropdownToggle caret size="lg" color={themeClass} />

          <DropdownMenu>

            <DropdownItem onClick={e => **this**.chooseTheme('Primary', e)}>Primary Theme</DropdownItem>

            <DropdownItem onClick={e => **this**.chooseTheme('Danger', e)}>Danger Theme</DropdownItem>

            <DropdownItem onClick={e => **this**.chooseTheme('Success', e)}>Success Theme</DropdownItem>

            <DropdownItem divider />

          <DropdownItem onClick={**this**.resetTheme}>Default Theme</DropdownItem>

          </DropdownMenu>

        </ButtonDropdown>

      </div>

    );

  }

}

export **default** ThemeSwitcher;

Now, update the **src/index.js** file with the following snippet.

**Index.js**

**import** 'bootstrap/dist/css/bootstrap.min.css';

**import** React from 'react';

**import** ReactDOM from 'react-dom';

**import** App from './App.js';

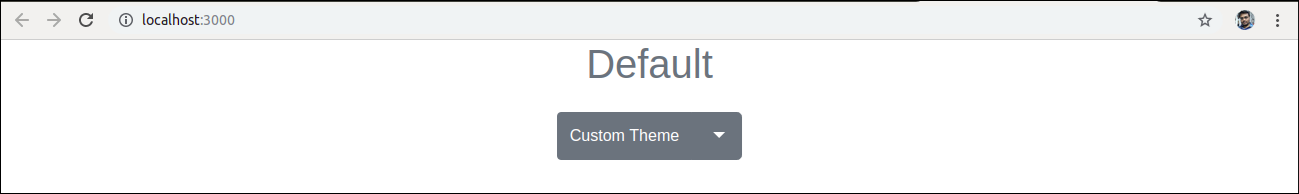
**import** './index.css';

**import** ThemeSwitcher from './ThemeSwitcher';

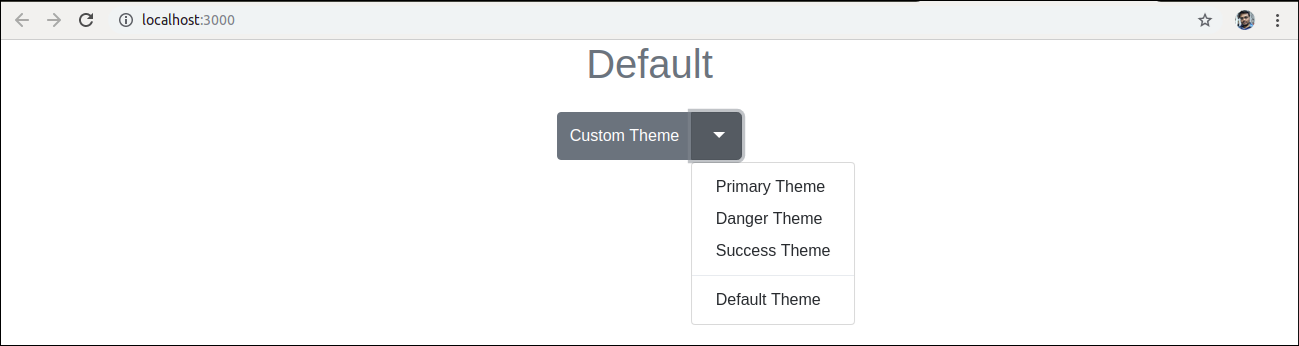
ReactDOM.render(<ThemeSwitcher />, document.getElementById('root'));

**Output**

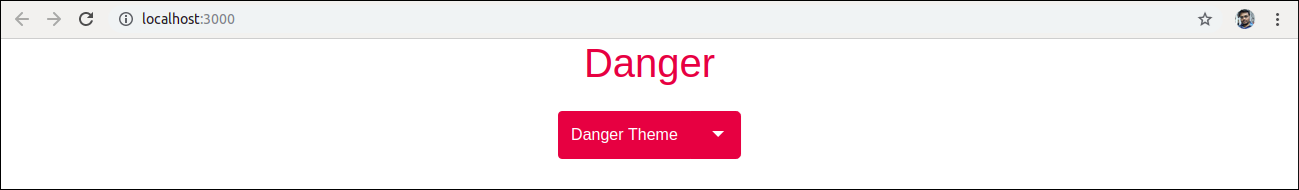
When we execute the React app, we should get the output as below.



Click on the dropdown menu. We will get the following screen.



Now, if we choose the **Danger Theme**, we will get the below screen.



# **React Map**

A map is a data collection type where data is stored in the form of pairs. It contains a unique key. The value stored in the map must be mapped to the key. We cannot store a duplicate pair in the map(). It is because of the uniqueness of each stored key. It is mainly used for fast searching and looking up data.

In React, the ?map? method used to traverse and display a list of similar objects of a component. A map is not the feature of React. Instead, it is the standard JavaScript function that could be called on any array. The map() method creates a new array by calling a provided function on every element in the calling array.

### **Example**

In the given example, the map() function takes an array of numbers and double their values. We assign the new array returned by map() to the variable doubleValue and log it.

var numbers = [1, 2, 3, 4, 5];

**const** doubleValue = numbers.map((number)=>{

**return** (number \* 2);

});

console.log(doubleValue);

## In React, the map() method used for:

1. Traversing the list element.

**Example**

**import** React from 'react';

**import** ReactDOM from 'react-dom';

function NameList(props) {

**const** myLists = props.myLists;

**const** listItems = myLists.map((myList) =>

    <li>{myList}</li>

  );

**return** (

    <div>

          <h2>React Map Example</h2>

              <ul>{listItems}</ul>

    </div>

  );

}

**const** myLists = ['A', 'B', 'C', 'D', 'D'];

ReactDOM.render(

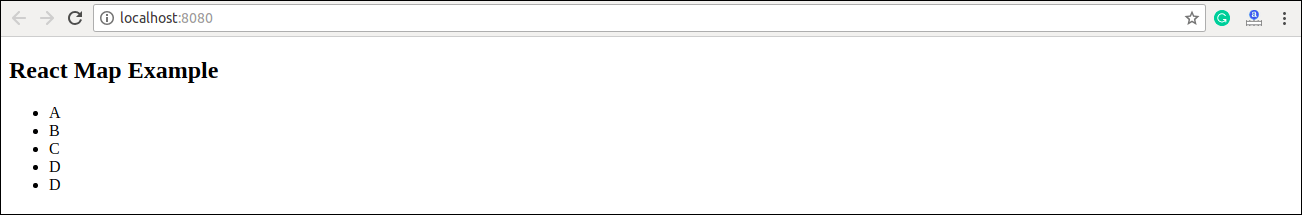
  <NameList myLists={myLists} />,

  document.getElementById('app')

);

export **default** App;

**Output**



2. Traversing the list element with keys.

**Example**

**import** React from 'react';

**import** ReactDOM from 'react-dom';

function ListItem(props) {

**return** <li>{props.value}</li>;

}

function NumberList(props) {

**const** numbers = props.numbers;

**const** listItems = numbers.map((number) =>

    <ListItem key={number.toString()}

              value={number} />

  );

**return** (

    <div>

      <h2>React Map Example</h2>

          <ul> {listItems} </ul>

    </div>

  );

}

**const** numbers = [1, 2, 3, 4, 5];

ReactDOM.render(

  <NumberList numbers={numbers} />,

  document.getElementById('app')

);

export **default** App;

**Output**

