

ReactJS

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.
- React is a library for building composable user interfaces. It encourages the creation of reusable UI components, components, which present data that changes over time. Lots of people use React as the V in MVC. React abstracts away the DOM from you, offering a simpler programming model and better performance. React can also render on the server using Node, and it can power native apps using React Native. React implements one-way reactive data flow, which reduces the boilerplate and is easier to reason about than traditional data binding.
- 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 Environment Setup

Pre-requisite for ReactJS

- NodeJS and NPM or **YARN** - ?
- React and React DOM
- Webpack
- Babel

Ways to install ReactJS

There are two ways to set up an environment for successful ReactJS application. They are given below.

- Using the npm command
- **Using the create-react-app command**

React Environment Setup

YARN

An alternative to npm is Yarn. It was released in 2016 by Facebook in collaboration with Exponent, Google, and Tilde. The project helps Facebook and other companies manage their dependencies reliably. If you're familiar with the npm workflow, getting up to speed with Yarn is fairly simple. First, install Yarn globally with npm:

- `npm install -g yarn`

Then, you're ready to install packages. When installing dependencies from package.json, in place of npm install, you can run yarn.

- To install a specific package with yarn, run:

`yarn add package-name`

- To remove a dependency, the command is familiar, too:

`yarn remove package-name`

Yarn is used in production by Facebook and is included in projects like React, React Native, and Create React App. If you ever find a project that contains a yarn.lock file, the project uses yarn. Similar to the npm install command, you can install all the dependencies of the project by typing yarn.

NPX:

The npx stands for Node Package Execute and it comes with the npm, when you installed npm above 5.2.0 version then automatically npx will installed. It is an npm package runner that can execute any package that you want from the npm registry without even installing that package. The npx is useful during a single time use package. If you have installed npm below 5.2.0 then npx is not installed in your system. You can check npx is installed or not by running the following command:

`npx -v`

If npx is not installed you can install that separately by running the below command.

`npm install -g npx`

Execute package with npx:

Directly runnable: You can execute your package without installation, to do so run the following command.

`npx your-package-name`

React Environment Setup

2. Using the create-react-app command

If you do not want to install react by using webpack and babel, then you can choose create-react-app to install react. The 'create-react-app' is a tool maintained by Facebook itself. This is suitable for beginners without manually having to deal with transpiling tools like webpack and babel. In this section,

Install NodeJS and NPM

NodeJS and NPM are the platforms need to develop any ReactJS application. You can install NodeJS and NPM package manager by the link given below.

Install React

You can install React using npm package manager by using the below command. There is no need to worry about the complexity of React installation. The create-react-app npm package will take care of it.

```
npm install -g create-react-app
```

Create a new React project

After the installation of React, you can create a new react project using create-react-app command. Here, I choose **myapp** name for my project.

```
create-react-app myapp
```

If ExecutionPolicy issue:

```
set-ExecutionPolicy RemoteSigned -Scope CurrentUser
```

React Environment Setup

React create-react-app

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

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

Requirements

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

- Node version ≥ 8.10
- NPM version ≥ 5.6

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

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

React Features

Features:

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

JSX

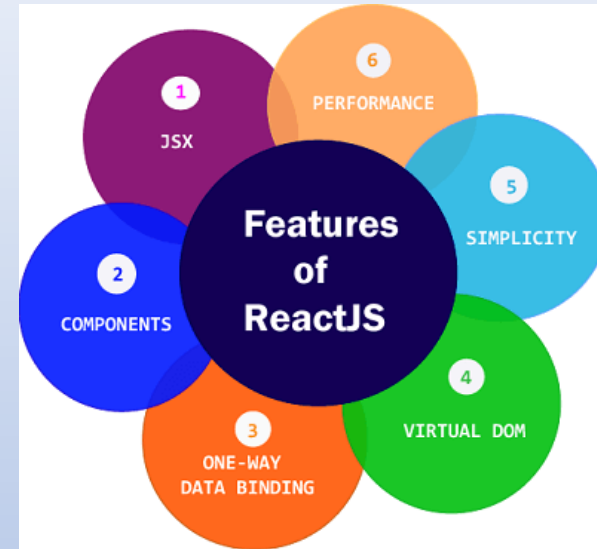
JSX stands for JavaScript XML. It is a JavaScript syntax extension. Its an XML or HTML like syntax used by ReactJS. This syntax is processed into JavaScript calls of React Framework. It extends the ES6 so that HTML like text can co-exist with JavaScript react code. It is not necessary to use JSX, but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



React Features

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. Virtual DOM exists which is like a lightweight copy of the actual DOM. So for every object that exists in the original DOM, there is an object for that in React Virtual DOM. It is exactly the same, but it does not have the power to directly change the layout of the document. Manipulating DOM is slow, but manipulating Virtual DOM is fast as nothing gets drawn on the screen.

Simplicity

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

Performance

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

Example:

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

var name = "Learner";

var element = <h1>Hello, { name }.Welcome to MyInstitute.</h1>;

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


React Features

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 Dev [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.

React Features

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.

Difference Between AngularJS and ReactJS

AngularJS

AngularJS is an open-source JavaScript framework used to build a dynamic web application. Misko Hevery and Adam Abrons developed AngularJS in 2009, and now Google maintained it. The latest version of Angular is 1.7.8 on March 11, 2019. It is based on HTML and JavaScript and mostly used for building a Single Page Application. It can be included to an HTML page with a `<script>` tag. It extends HTML by adding built-in attributes with the directive and binds data to HTML with Expressions.

Features of AngularJS

1. **Data-binding:** AngularJS follows the two-way data binding. It is the automatic synchronization of data between model and view components.
2. **POJO Model:** AngularJS uses POJO (Plain Old JavaScript) model, which provides spontaneous and well-planned objects. The POJO model makes AngularJS self-sufficient and easy to use.
3. **Model View Controller(MVC) Framework:** MVC is a software design pattern used for developing web applications. The working model of AngularJS is based on MVC patterns. The MVC Architecture in AngularJS is easy, versatile, and dynamic. MVC makes it easier to build a separate client-side application.
4. **Services:** AngularJS has several built-in services such as `$http` to make an XMLHttpRequest.
5. **User interface with HTML:** In AngularJS, User interfaces are built on HTML. It is a declarative language which has shorter tags and easy to comprehend. It provides an organized, smooth, and structured interface.
6. **Dependency Injection:** AngularJS has a built-in dependency injection subsystem that helps the developer to create, understand, and test the applications easily.
7. **Active community on Google:** AngularJS provides excellent community support. It is Because Google maintains AngularJS. So, if you have any maintenance issues, there are many forums available where you can get your queries solved.
8. **Routing:** Routing is the transition from one view to another view. Routing is the key aspect of single page applications where everything comes in a single page. Here, developers do not want to redirect the users to a new page every time they click the menu. The developers want the content load on the same page with the URL changing.

Difference Between AngularJS and ReactJS

ReactJS

ReactJS is an open-source JavaScript library used to build a user interface for Single Page Application. It is responsible only for the view layer of the application. It provides developers to compose complex UIs from a small and isolated piece of code called "components." ReactJS made of two parts first is components, that are the pieces that contain HTML code and what you want to see in the user interface, and the second one is HTML document where all your components will be rendered.

Jordan Walke, who was a software engineer at Facebook, develops it. Initially, it was developed and maintained by Facebook and was later used in its products like WhatsApp & Instagram. Facebook developed ReactJS in 2011 for the newsfeed section, but it was released to the public in May 2013.

Features of ReactJS

1. **JSX:** JSX is a JavaScript syntax extension. The JSX 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.
2. **Components:** ReactJS is all about components. ReactJS application is made up of multiple components, and each component has its logic and controls. These components can be reusable, which help you to maintain the code when working on larger scale projects.
3. **One-way Data Binding:** ReactJS follows unidirectional data flow or one-way data binding. The one-way data binding gives 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.
4. **Virtual DOM:** A virtual DOM object is a representation of the real DOM object. 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 are changed. It makes the application faster, and there is no wastage of memory.
5. **Simplicity:** ReactJS uses the JSX file, which makes the application simple and to code as well as understand. Also, ReactJS is a component-based approach which makes the code reusable as your need. It makes it simple to use and learn.
6. **Performance:** ReactJS is known to be a great performer. The reason behind this is that it manages a virtual DOM. 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.

Difference Between AngularJS and ReactJS

Consolidated difference:

	AngularJS	ReactJS
Author	Google	Facebook Community
Developer	Misko Hevery	Jordan Walke
Initial Release	October 2010	March 2013
Latest Version	Angular 1.7.8 on 11 March 2019.	React 16.8.6 on 27 March 2019
Language	JavaScript, HTML	JSX
Type	Open Source MVC Framework	Open Source JS Framework
Rendering	Client-Side	Server-Side
Packaging	Weak	Strong
Data-Binding	Bi-directional	Uni-directional
DOM	Regular DOM	Virtual DOM
Testing	Unit and Integration Testing	Unit Testing
App Architecture	MVC	Flux
Dependencies	It manages dependencies automatically.	It requires additional tools to manage dependencies.
Routing	It requires a template or controller to its router configuration, which has to be managed manually.	It doesn't handle routing but has a lot of modules for routing, eg., react-router.
Performance	Slow	Fast, due to virtual DOM.
Best For	It is best for single page applications that update a single view at a time.	It is best for single page applications that update multiple views at a time.

JavaScript for ReactJS

Since its release in 1995, JavaScript has gone through many changes. At first, we used JavaScript to add interactive elements to web pages: button clicks, hover states, form validation, etc.. Later, JavaScript got more robust with DHTML and AJAX. Today, with Node.js, JavaScript has become a real software language that's used to build full-stack applications. JavaScript is everywhere.

JavaScript's evolution has been guided by a group of individuals from companies that use JavaScript, browser vendors, and community leaders. The committee in charge of shepherding the changes to JavaScript over the years is the European Computer Manufacturers Association (ECMA). Changes to the language are community-driven, originating from proposals written by community members. Anyone can submit a proposal to the ECMA committee. The responsibility of the ECMA committee is to manage and prioritize these proposals to decide what's included in each spec.

The first release of ECMAScript was in 1997, ECMAScript1. This was followed in 1998 by ECMAScript2. ECMAScript3 came out in 1999, adding regular expressions, string handling, and more. The process of agreeing on an ECMAScript4 became a chaotic, political mess that proved to be impossible. It was never released. In 2009, ECMAScript5(ES5) was released, bringing features like new array methods, object properties, and library support for JSON.

Since then, there has been a lot more momentum in this space. After ES6 or ES2015 was released in, yes, 2015, there have been yearly releases of new JS features. Anything that's part of the stage proposals is typically called ESNext, which is a simplified way of saying this is the next stuff that will be part of the JavaScript spec

Proposals are taken through clearly defined stages, from stage 0, which represents the newest proposals, up through stage 4, which represents the finished proposals. When a proposal gains traction, it's up to the browser vendors like Chrome and Firefox to implement the features. Consider the `const` keyword. When creating variables, we used to use `var` in all cases. The ECMA committee decided there should be a `const` keyword to declare constants (more on that later in the chapter). When `const` was first introduced, you couldn't just write `const` in JavaScript code and expect it to run in a browser. Now you can because browser vendors have changed the browser to support it.

Declaring Variables

Prior to ES2015, the only way to declare a variable was with the `var` keyword. We now have a few different options that provide improved functionality.

The `const` Keyword

A constant is a variable that cannot be overwritten. Once declared, you cannot change its value. A lot of the variables that we create in JavaScript should not be overwritten, so we'll be using `const` a lot. Like other languages had done before it, JavaScript introduced constants with ES6.

Before constants, all we had were variables, and variables could be overwritten:

```
var pizza = true;
pizza = false;
console.log(pizza); // false
```

We cannot reset the value of a constant variable, and it will generate a console error (as shown in Figure 2-1) if we try to overwrite the value:

ReactJS

Let's say one of your friends posted a photograph on Facebook. Now you go and like the image and then you started checking out the comments too. Now while you are browsing over comments you see that the likes count has increased by 100, since you liked the picture, even without reloading the page. This magical count change is because of Reactjs. React is a declarative, efficient, and flexible JavaScript library for building user interfaces. It's 'V' in MVC. ReactJS is an open-source, component-based front end library responsible only for the view layer of the application. It is maintained by Facebook.

React uses a declarative paradigm that makes it easier to reason about your application and aims to be both efficient and flexible. It designs simple views for each state in your application, and React will efficiently update and render just the right component when your data changes. The declarative view makes your code more predictable and easier to debug.

A React application is made of multiple components, each responsible for rendering a small, reusable piece of HTML. Components can be nested within other components to allow complex applications to be built out of simple building blocks. A component may also maintain an internal state – for example, a TabList component may store a variable corresponding to the currently open tab.

Note: React is not a framework. It is just a library developed by Facebook to solve some problems that we were facing earlier.

Example: Create a new React project by using the command below:

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

- Filename App.js: Now change the App.js file with the given below code:

```
import React,{ Component } from 'react';
class App extends Component {
  render() {
    return (
      <div>
        <h1>Hello, Learner.Welcome to MyInstitute.</h1>
      </div>
    );
  }
}
export default App;
```

ReactJS

How does it work: While building client-side apps, a team of Facebook developers realized that the DOM is slow (The Document Object Model (DOM) is an application programming interface (API) for HTML and XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated.). So, to make it faster, React implements a virtual DOM that is basically a DOM tree representation in JavaScript. So when it needs to read or write to the DOM, it will use the virtual representation of it. Then the virtual DOM will try to find the most efficient way to update the browser's DOM.

Unlike browser DOM elements, React elements are plain objects and are cheap to create. React DOM takes care of updating the DOM to match the React elements. The reason for this is that JavaScript is very fast and it's worth keeping a DOM tree in it to speed up its manipulation.

Although React was conceived to be used in the browser, because of its design it can also be used in the server with Node.js.

React JSX

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

JSX File - `<div>Hello JSX</div>`

Corresponding Output - `React.createElement("div", null, "Hello JSX");`

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 JSX."

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.

React JSX

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>MyInstitute</h1>

        <h2>Training Institutes</h2>

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

      </div>

    );

  }

}

export default App;
```

React JSX Attribute

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>MyInstitute</h1>
        <h2>Training Institutes</h2>
        <p data-demoAttribute = "demo">This website contains the best CS tutorials.</p>
      </div>
    );
  }
}

export default App;
```

React JSX Attribute

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

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

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

Example

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

class App extends Component{

  render(){

    return(

      <div>

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

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

      </div>

    );

  }

}

export default App;
```

React JSX Attribute

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

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

Example

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

```
class App extends Component{
```

```
  render(){
```

```
    return(
```

```
      <div>
```

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

```
      </div>
```

```
    );
```

```
  }
```

```
}
```

```
export default App;
```

React JSX Comments

JSX Comments

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

Example

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

class App extends Component{
  render(){
    return(
      <div>
        <h1 className = "hello" >Hello MyInstitute</h1>
        { /* This is a comment in JSX */ }
      </div>
    );
  }
}

export default App;
```

React JSX Styling

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.MyInstitute.com</h1>
      </div>
    );
  }
}
export default App;
```

React JSX Styling

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 i = 5;

    return (
      <div>
        <h1>{i == 1 ? 'True!' : 'False!'}</h1>
      </div>
    );
  }
}

export default App;
```


React DOM

What is DOM?

DOM, abbreviated as Document Object Model, is a World Wide Web Consortium standard logical representation of any webpage. In easier words, DOM is a tree-like structure that contains all the elements and its properties of a website as its nodes. DOM provides a language-neutral interface that allows accessing and updating of the content of any element of a webpage.

Before React, Developers directly manipulated the DOM elements which resulted in frequent DOM manipulation, and each time an update was made the browser had to recalculate and repaint the whole view according to the particular CSS of the page, which made the total process to consume a lot of time. As a betterment, React brought into the scene the virtual DOM. The **Virtual DOM** can be referred to as a copy of the actual DOM representation that is used to hold the updates made by the user and finally reflect it over to the original Browser DOM at once consuming much lesser time.

What is ReactDOM?

ReactDOM is a package that provides DOM specific methods that can be used at the top level of a web app to enable an efficient way of managing DOM elements of the web page. ReactDOM provides the developers with an API containing the following methods and a few more.

```
render()
ReactDOMNode()
unmountComponentAtNode()
hydrate()
createPortal()
```

Pre-requisite: To use the ReactDOM in any React web app we must first import ReactDOM from the react-dom package by using the following code snippet:

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

React DOM

render() Function

This is one of the most important methods of ReactDOM. This function is used to render a single React Component or several Components wrapped together in a Component or a div element. This function uses the efficient methods of React for updating the DOM by being able to change only a subtree, efficient diff methods, etc.

Syntax:

`ReactDOM.render(element, container, callback)`

Parameters: This method can take a maximum of three parameters as described below.

element: This parameter expects a JSX expression or a React Element to be rendered.

container: This parameter expects the container in which the element has to be rendered.

callback: This is an optional parameter that expects a function that is to be executed once the render is complete.

Return Type: This function returns a reference to the component or null if a stateless component was rendered.

findDOMNode() Function This function is generally used to get the DOM node where a particular React component was rendered. This method is very less used like the following can be done by adding a ref attribute to each component itself.

Syntax: : `ReactDOM.findDOMNode(component)`

Parameters: This method takes a single parameter component that expects a React Component to be searched in the Browser DOM.

Return Type: This function returns the DOM node where the component was rendered on success otherwise null.

React DOM

unmountComponentAtNode() Function

This function is used to unmount or remove the React Component that was rendered to a particular container. As an example, you may think of a notification component, after a brief amount of time it is better to remove the component making the web page more efficient.

Syntax: ReactDOM.unmountComponentAtNode(container)

Parameters: This method takes a single parameter container which expects the DOM container from which the React component has to be removed.

Return Type: This function returns true on success otherwise false.

hydrate() Function

This method is equivalent to the render() method but is implemented while using server-side rendering.

Syntax: ReactDOM.hydrate(element, container, callback)

Parameters: This method can take a maximum of three parameters as described below.

element: This parameter expects a JSX expression or a React Component to be rendered.

container: This parameter expects the container in which the element has to be rendered.

callback: This is an optional parameter that expects a function that is to be executed once the render is complete.

Return Type: This function attempts to attach event listeners to the existing markup and returns a reference to the component or null if a stateless component was rendered.

React DOM

createPortal() Function

Usually, when an element is returned from a component's render method, it's mounted on the DOM as a child of the nearest parent node which in some cases may not be desired. Portals allow us to render a component into a DOM node that resides outside the current DOM hierarchy of the parent component.

Syntax: ReactDOM.createPortal(child, container)

Parameters: This method takes two parameters as described below.

child: This parameter expects a JSX expression or a React Component to be rendered.

container: This parameter expects the container in which the element has to be rendered.

Return Type: This function returns nothing.

Important Points to Note:

ReactDOM.render() replaces the child of the given container if any. It uses a highly efficient diff algorithm and can modify any subtree of the DOM.

findDOMNode() function can only be implemented upon mounted components thus Functional components can not be used in findDOMNode() method.

ReactDOM uses observables thus provides an efficient way of DOM handling.

ReactDOM can be used on both the client-side and server-side.

Virtual DOM

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

Updating DOM: If you know a little about JavaScript then you might have seen people making use of 'getElementById()' or 'getElementsByClass()' method to modify the content of DOM. Every time there is a change in the state of your application, the DOM gets updated to reflect that change in the UI. Though doing things like this is not a problem and it works fine, but consider a case where we have a DOM that contains nodes in a large number, and also all these web elements have different styling and attributes. As DOM is represented as a tree itself, updating the tree here is not a costly operation indeed we have a lot of algorithms on trees to make the updates fast. What's proving to be costly is everytime the DOM gets updated, the updated element and its children have to be rendered again to update the UI of our page. Like this each time there is a component update, the DOM needs to be updated and the UI components have to be re-rendered.

Example:

```
// Simple getElementById() method
```

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

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

- The browser parses the HTML to find the node with this id.
- It removes the child element of this specific element.
- Updates the element(DOM) with the 'updated value'.
- Recalculates the CSS for the parent and child nodes.
- Update the layout.

Finally, traverse the tree and paint it on the screen(browser) display.

So as we know now that updating the DOM not only involves changing the content, it has a lot more attached to it. Also recalculating the CSS and changing the layouts involves complex algorithms, and they do affect the performance. So React has a different approach to dealing with this, as it makes use of something known as Virtual DOM.

Virtual DOM

Virtual DOM: React uses Virtual DOM exists which is like a lightweight copy of the actual DOM(a virtual representation of the DOM). So for every object that exists in the original DOM, there is an object for that in React Virtual DOM. It is exactly the same, but it does not have the power to directly change the layout of the document. Manipulating DOM is slow, but manipulating Virtual DOM is fast as nothing gets drawn on the screen. So each time there is a change in the state of our application, virtual DOM gets updated first instead of the real DOM. You may still wonder, “Aren’t we doing the same thing again and doubling our work? How can this be faster?” Read below to understand how things will be faster using virtual DOM.

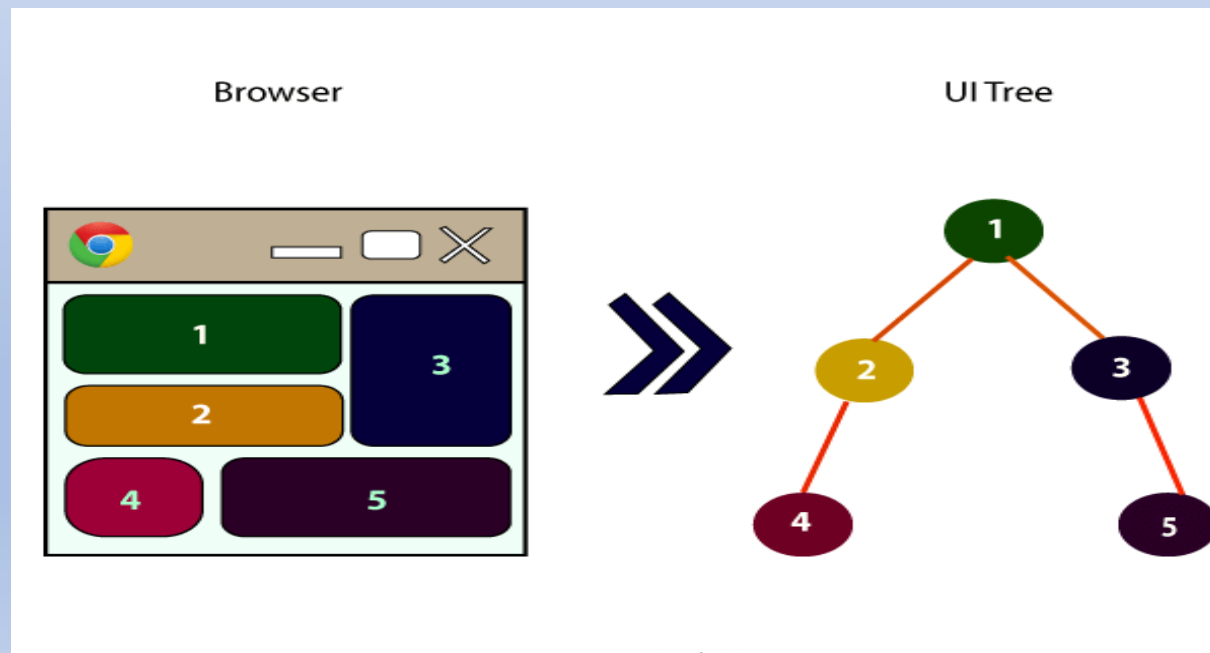
How Virtual DOM actually make the things faster: When anything new is added to the application, a virtual DOM is created and it is represented as a tree. Each element in the application is a node in this tree. So, whenever there is a change in state of any element, a new Virtual DOM tree is created.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

How Virtual DOM helps React: In react, everything is treated as a component be it a functional component or class component. A component can contain a state. Each time we change something in our JSX file or let’s put it in simple terms, whenever the state of any component is changed react updates it’s Virtual DOM tree. Though it may sound that it is ineffective but the cost is not much significant as updating the virtual DOM doesn’t take much time. React maintains two Virtual DOM at each time, one contains the updated Virtual DOM and one which is just the pre-update version of this updated Virtual DOM. Now it compares the pre-update version with the updated Virtual DOM and figures out what exactly has changed in the DOM like which components have been changed. This process of comparing the current Virtual DOM tree with the previous one is known as **‘diffing’**. Once React finds out what exactly has changed then it updated those objects only, on real DOM. React uses something called as batch updates to update the real DOM. It just mean that the changes to the real DOM are sent in batches instead of sending any update for a single change in the state of a component. We have seen that the re-rendering of the UI is the most expensive part and React manages to do this most efficiently by ensuring that the Real DOM receives batch updates to re-render the UI. This entire proces of transforming changes to the real DOM is called **Reconciliation**

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

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.



React Components

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

1. Functional Components
2. Class Components

Functional Components

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

```
function WelcomeMessage(props) {  
  return <h1>Welcome to the , {props.name}</h1>;  
}
```

The functional component is also known as a stateless component because they do not hold or manage state. 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>  
    );  
  }  
}
```


React Components

```
class First extends React.Component {  
  render() {  
    return (  
      <div>  
        <h1>MyInstitute</h1>  
      </div>  
    );  
  }  
}  
class Second extends React.Component {  
  render() {  
    return (  
      <div>  
        <h2>www.MyInstitute.com</h2>  
        <p>This websites contains the great CS tutorial.</p>  
      </div>  
    );  
  }  
}  
export default App;
```

React Components

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.

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

```
class App extends React.Component {
```

```
  constructor() {  
    super();  
    this.state = {  
      data:  
      [  
        {  
          "name": "Abhishek"  
        },  
        {  
          "name": "Saharsh"  
        },  
        {  
          "name": "Ajay"  
        }  
      ]  
    }  
  }  
}
```

React Components

```
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;
```

React Components

Functional Components vs Class Components:

Functional Components	Class Components
A functional component is just a plain JavaScript function that accepts props as an argument and returns a React element.	A class component requires you to extend from <code>React.Component</code> and create a render function which returns a React element.
There is no <code>render</code> method used in functional components.	It must have the <code>render()</code> method returning HTML
Also known as Stateless components as they simply accept data and display them in some form, that they are mainly responsible for rendering UI.	Also known as Stateful components because they implement logic and state.
React lifecycle methods (for example, <code>componentDidMount</code>) cannot be used in functional components.	React lifecycle methods can be used inside class components (for example, <code>componentDidMount</code>).

Pure Components

Generally, In ReactJS, we use `shouldComponentUpdate()` Lifecycle method to customize the default behavior and implement it when the React component should re-render or update itself.

Now, ReactJS has provided us a Pure Component. If we extend a class with Pure Component, there is no need for `shouldComponentUpdate()` Lifecycle Method. ReactJS Pure Component Class compares current state and props with new props and states to decide whether the React component should re-render itself or Not.

In simple words, If the previous value of state or props and the new value of state or props is the same, the component will not re-render itself. Since Pure Components restricts the re-rendering when there is no use of re-rendering of the component. Pure Components are Class Components which extends `React.PureComponent`.

```
import React from 'react';

export default class Test extends React.PureComponent{
  render(){
    return <h1>Welcome to MyInstitute</h1>;
  }
}
```

Extending React Class Components with Pure Components ensures the higher performance of the Component and ultimately makes your application faster, While in the case of Regular Component, it will always re-render either value of State and Props changes or not.

While using Pure Components, Things to be noted are that, In these components, the Value of State and Props are Shallow Compared (Shallow Comparison) and It also takes care of “`shouldComponentUpdate`” Lifecycle method implicitly.

So there is a possibility that if these State and Props Objects contain nested data structure then Pure Component’s implemented `shouldComponentUpdate` will return false and will not update the whole subtree of Children of this Class Component. So in Pure Component, the nested data structure doesn’t work properly.

In this case, State and Props Objects should be simple objects and Child Elements should also be Pure, means to return the same output for the same input values at any instance.

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.

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.

React State

Example:

```
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>MyInstitute 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 MyInstitute!! </h1>
        { bio }
      </div>
    );
  }
}
export default App;
```

React State

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.

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


React State

```
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 MyInstitute!!</h1>
        {
          this.state.displayBio ? (
            <div>
              <p><h4>MyInstitute is one of the best Java training institute in Noida, Delhi, Gurugram, Ghaziabad and Faridabad. We have a team of experienced Java developers a
nd 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>
    )
  }
}
```

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

```
      </div>
```

```
    );
```

```
  }
```

```
}
```

```
export default App;
```

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> MyInstitute 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 = "MyInstitute!!" />, document.getElementById('app'));
```

React Props

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.

Example:

App.js

```
import React, { Component } from 'react';
class App extends React.Component {
  render() {
    return (
      <div>
        <h1>Default Props Example</h1>
        <h3>Welcome to {this.props.name}</h3>
        <p>MyInstitute is one of the best Java training institute in Noida, Delhi, Gurugram, Ghaziabad and Faridabad.</p>
      </div>
    );
  }
}
App.defaultProps = {
  name: "MyInstitute"
}
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'));
```

React Props

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.

Example:

App.js

```
import React, { Component } from 'react';
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: "MyInstitute",
    }
  }
  render() {
    return (
      <div>
        <JTP jtpProp = {this.state.name}/>
      </div>
    );
  }
}
```

React Props

```
class JTP extends React.Component {  
  render() {  
    return (  
      <div>  
        <h1>State & Props Example</h1>  
        <h3>Welcome to {this.props.jtpProp}</h3>  
        <p>MyInstitute 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'));
```

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.

```
class App extends React.Component {  
  render() {}  
}  
  
Component.propTypes = { /*Definition */};
```

React Props Validation

ReactJS Props Validator

ReactJS props validator contains the following list of validators.

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

React Props Validation

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.

```
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>
```

React Props Validation

```
<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: "MyInstitute",
}
export default App;
```

React Props Validation

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.

State Vs. Props

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

Note: The component State and Props share some common similarities. They are given in the below table.

SN	State and Props
1.	Both are plain JS object.
2.	Both can contain default values.
3.	Both are read-only when they are using by this.

React Constructor

What is Constructor?

The constructor is a method used to initialize an object's state in a class. It is 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.

```
Constructor(props){  
  super(props);  
}
```

In React, constructors are mainly used for two purposes:

It is used for initializing the local state of the component by assigning an object to this.state.

It is used for binding event handler methods that occur in your component.

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

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

React Constructor

Example:

App.js:

```
import React, { Component } from 'react';
class App extends Component {
  constructor(props){
    super(props);
    this.state = {
      data: 'www.MyInstitute.com'
    }
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick(){
    console.log(this.props);
  }
  render() {
    return (
      <div className="App">
        <h2>React Constructor Example</h2>
        <input type="text" value={this.state.data} />
        <button onClick={this.handleClick}>Please Click</button>
      </div>
    );
  }
}
export default App;
```

React Constructor

Example:

App.js:

```
import React, { Component } from 'react';
class App extends Component {
  constructor(props){
    super(props);
    this.state = {
      data: 'www.MyInstitute.com'
    }
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick(){
    console.log(this.props);
  }
  render() {
    return (
      <div className="App">
        <h2>React Constructor Example</h2>
        <input type="text" value={this.state.data} />
        <button onClick={this.handleClick}>Please Click</button>
      </div>
    );
  }
}
export default App;
```


React Constructor

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).

React Constructor

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.MyInstitute.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;

React Constructor

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);  
  }  
}
```

React Constructor

We can use a constructor in the following ways:

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.

`this.setState(object newState[, function callback]);`

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

React Component API

Example:

```
import React, { Component } from 'react';
import PropTypes from 'prop-types';
class App extends React.Component {
  constructor() {
    super();
    this.state = {
      msg: "Welcome to MyInstitute"
    };
    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;
```

React Component API

forceUpdate() - This method allows us to update the component manually.

Syntax: Component.forceUpdate(callback);

```
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;
```

React Component API

forceUpdate() - This method allows us to update the component manually.

Syntax: Component.forceUpdate(callback);

Example:

```
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;
```

Each time when you click on **ForceUpdate** button, it will generate the **random** number.

React Component API

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

Syntax: ReactDOM.findDOMNode(component);

Example:

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

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>

); } }

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.

React Component Life-Cycle

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.

componentWillReceiveProps()

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.

React Component Life-Cycle

Example:

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

```
class App extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {hello: "MyInstitute"};  
    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>  
    );  
  }  
}
```

React Component Life-Cycle

Example:

```
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;
```

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.

- Uncontrolled component
- 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.

React Forms

Example: the code accepts a field **username** and **company name** in an uncontrolled component.

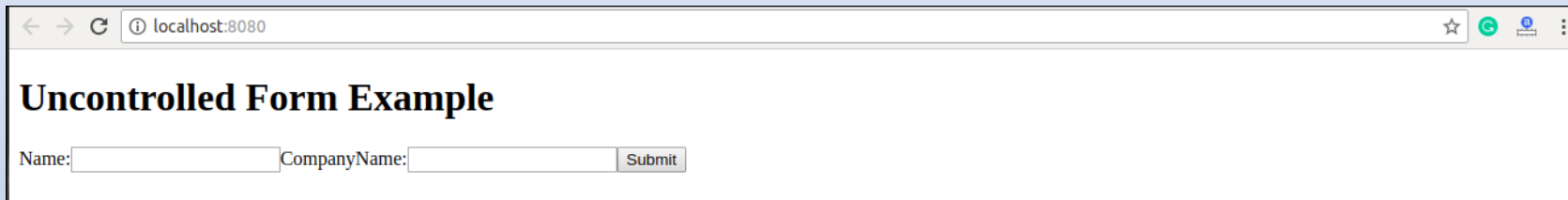
```
import React, { Component } from 'react';
class App extends React.Component {
  constructor(props) {
    super(props);
    this.updateSubmit = this.updateSubmit.bind(this);
    this.input = React.createRef();
  }
  updateSubmit(event) {
    alert('You have entered the UserName and CompanyName successfully.');
```

event.preventDefault();

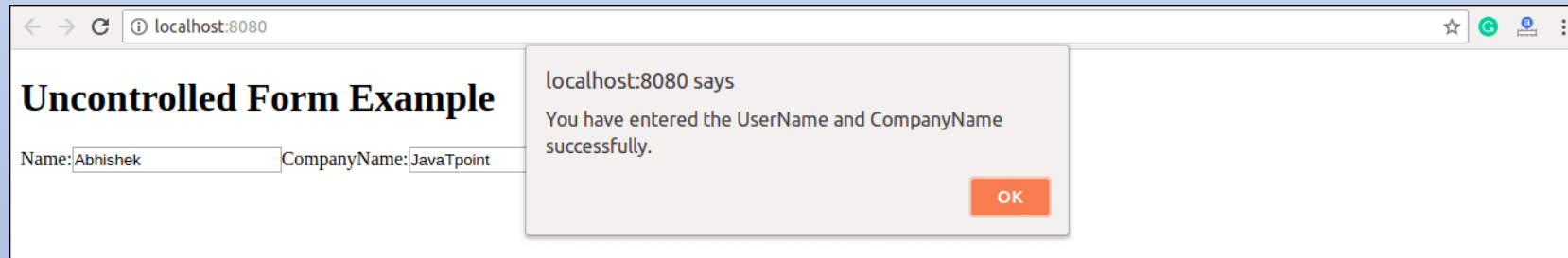
```
  }
  render() {
    return (
      <form onSubmit={this.updateSubmit}>
        <h1>Uncontrolled Form Example</h1>
        <label>Name:
          <input type="text" ref={this.input} />
        </label>
        <label>
          CompanyName:
          <input type="text" ref={this.input} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
export default App;
```

React Forms

Output:



A screenshot of a web browser window at localhost:8080. The page title is "Uncontrolled Form Example". Below the title, there is a form with two input fields: "Name:" and "CompanyName:". The "Name:" field is empty, and the "CompanyName:" field is also empty. To the right of the "CompanyName:" field is a "Submit" button.



A screenshot of a web browser window at localhost:8080. The page title is "Uncontrolled Form Example". Below the title, there is a form with two input fields: "Name:" and "CompanyName:". The "Name:" field contains the text "Abhishek" and the "CompanyName:" field contains the text "JavaTpoint". To the right of the "CompanyName:" field is a "Submit" button. A modal dialog box is displayed over the form, containing the text "localhost:8080 says" and "You have entered the UserName and CompanyName successfully." with an "OK" button.

Controller vs Uncontrolled Components

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();
  }
}
```

Controller vs Uncontrolled Components

```
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;
```

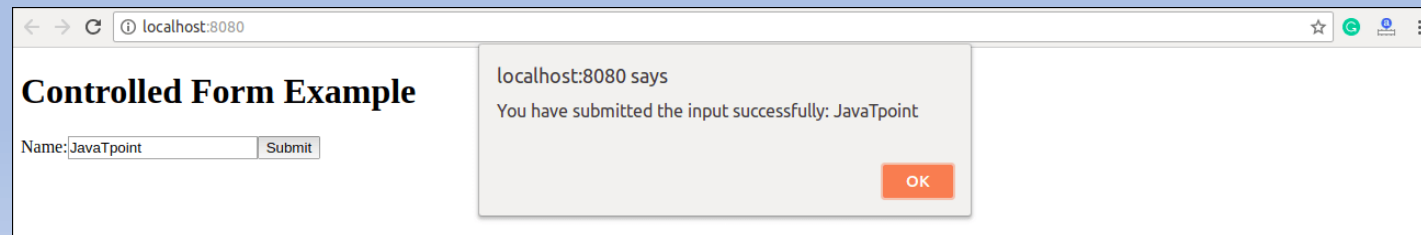
Output:



localhost:8080

Controlled Form Example

Name:



localhost:8080

Controlled Form Example

Name: JavaTpoint

localhost:8080 says
You have submitted the input successfully: JavaTpoint

Controller vs Uncontrolled Components

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.handleChange = this.handleChange.bind(this);
  }
  handleChange(event) {
    const target = event.target;
    const value = target.type === 'checkbox' ? target.checked : target.value;
    const name = target.name;
    this.setState({
      [name]: value
    });
  }
}
```

Controller vs Uncontrolled Components

```
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.handleChange} />  
        </label>  
      <br />  
      <label>  
        Number of persons:  
        <input  
          name="numberOfPersons"  
          type="number"  
          value={this.state.numberOfPersons}  
          onChange={this.handleChange} />  
        </label>  
      </form>  
    );  
  }  
}  
export default App;
```

Controller vs Uncontrolled Components

```
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.handleChange} />  
        </label>  
      <br />  
      <label>  
        Number of persons:  
        <input  
          name="numberOfPersons"  
          type="number"  
          value={this.state.numberOfPersons}  
          onChange={this.handleChange} />  
        </label>  
      </form>  
    );  
  }  
}  
export default App;
```

Controller vs Uncontrolled Components

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:

React events are named as **camelCase** instead of **lowercase**.

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 MyInstitute
</button>
```

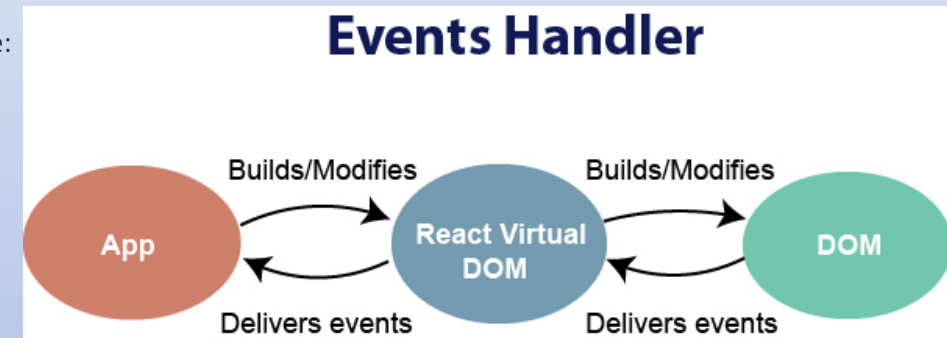
Event declaration in React:

```
<button onClick={showMessage}>
  Hello MyInstitute
</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>
```

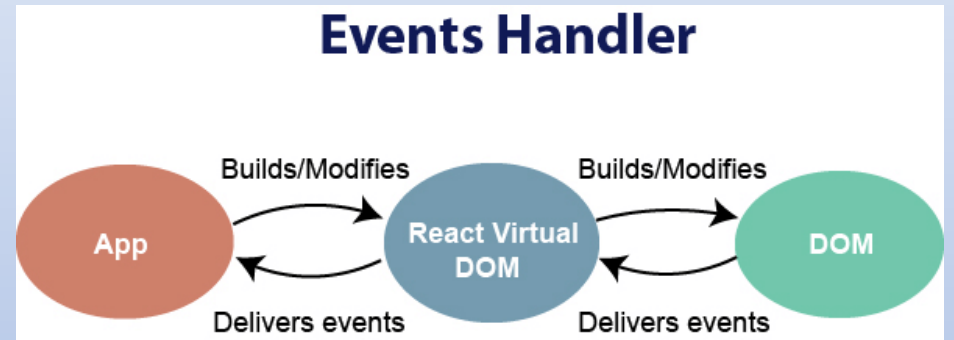


React Events

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.



React Events

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;

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

React Conditional Rendering

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')  
);
```

React Conditional Rendering

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.

```
('MyInstitute' == 'MyInstitute') && 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>
  );
}
```

React Conditional Rendering

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 MyInstitute':  
      return <Message text={text} />;  
    default:  
      return null;  
  }  
}
```

React Conditional Rendering

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 - 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**.

React Conditional Rendering

```
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};
  }
}
```

React Conditional Rendering

```
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>
  );
}
```


React Conditional Rendering

Preventing Component from 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')
);
```

React List

Lists are very useful when it comes to developing the UI of any website. Lists are mainly used for displaying menus in a website, for example, the navbar menu. In regular JavaScript, we can use arrays for creating lists. We can create lists in React in a similar manner as we do in regular JavaScript. We will see how to do this in detail further in this article. Let's first see how we can traverse and update any list in regular JavaScript. We can use the map() function in JavaScript for traversing the lists.

Below JavaScript code illustrate using map() function to traverse lists:

```
<script type="text/javascript">
  var numbers = [1,2,3,4,5];
  const updatedNums = numbers.map((number)=>{
    return (number + 2);
  });
  console.log(updatedNums);
</script>
```

Let us now create a list of elements in React. We will render the list numbers in the above code as an unordered list element in the browser rather than simply logging in to the console. To do this, we will traverse the list using the JavaScript map() function and updates elements to be enclosed between elements. Finally we will wrap this new list within elements and render it to the DOM.

```
import React from 'react';
import ReactDOM from 'react-dom';
const numbers = [1,2,3,4,5];
const updatedNums = numbers.map((number)=>{
  return <li>{number}</li>;
});
ReactDOM.render(
  <ul>
    {updatedNums}
  </ul>,
  document.getElementById('root')
);
```

React List

Rendering lists inside Components

In the above code in React, we had directly rendered the list to the DOM. But usually this is not a good practice to render lists in React. We already have talked about the uses of Components and had seen that everything in React is built as individual components. Consider the example of a Navigation Menu. It is obvious that in any website the items in a navigation menu are not hard coded. This item is fetched from the database and then displayed as lists in the browser. So from the component's point of view, we can say that we will pass a list to a component using props and then use this component to render the list to the DOM. We can update the above code in which we have directly rendered the list to now a component that will accept an array as props and returns an unordered list.

```
import React from 'react';
import ReactDOM from 'react-dom';
// Component that will return an unordered list
function Navmenu(props)
{
    const list = props.menuitems;

    const updatedList = list.map((listItems)=>{
        return <li>{listItems}</li>;
    });
    return(
        <ul>{updatedList}</ul>
    );
}
const menuItems = [1,2,3,4,5];
ReactDOM.render(
    <Navmenu menuitems = {menuItems} />,
    document.getElementById('root')
);
```

the unordered list is successfully rendered to the browser but a warning message is logged to the console.

Warning: Each child in an array or iterator
should have a unique "key" prop

React List

The above warning message says that each of the list items in our unordered list should have a unique key. A “key” is a special string attribute you need to include when creating lists of elements in React. We will discuss about keys in detail in further articles. For now, let’s just assign a string key to each of our list items in the above code.

Below is the updated code with keys:

```
import React from 'react';
import ReactDOM from 'react-dom';
// Component that will return an unordered list
function Navmenu(props)
{
  const list = props.menuitems;
  const updatedList = list.map((listItems)=>{
    return(
      <li key={listItems.toString()}>
        {listItems}
      </li>
    );
  });
  return(
    <ul>{updatedList}</ul>
  );
}
const menuitems = [1,2,3,4,5];
ReactDOM.render(
  <Navmenu menuitems = {menuitems} />,
  document.getElementById('root')
);
```

This code will give the same output as that of the previous code but this time without any warning. Keys are used in React to identify which items in the list are changed, updated, or deleted. In other words, we can say that keys are used to give an identity to the elements in the lists.

React Keys

A “key” is a special string attribute you need to include when creating lists of elements in React. Keys are used to React to identify which items in the list are changed, updated, or deleted. In other words, we can say that keys are used to give an identity to the elements in the lists. The next thing that comes to mind is that what should be good to be chosen as key for the items in lists. It is recommended to use a string as a key that uniquely identifies the items in the list. Below example shows a list with string keys:

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

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

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

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

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

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

React Keys

Using Keys with Components

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

Below code shows **incorrect usage** of keys:

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

// Component to be extracted
function MenuItem(props)
{
    const item = props.item;

    return(
        <li>
            {item}
        </li>
    );
}

// Component that will return an
// unordered list
```

React Keys

```
function Navmenu(props)
{
    const list = props.menuitems;

    const updatedList = list.map((listItems)=>{
        return (

        );
    });

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

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

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

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

React Keys

```
import React from 'react';
import ReactDOM from 'react-dom';
// Component to be extracted
function MenuItem(props)
{
    const item = props.item;

    return(
        <li>
            {item}
        </li>
    );
}
// Component that will return an
// unordered list
function Navmenu(props)
{
    const list = props.menuitems;
    const updatedList = list.map((listItems)=>{
        return (
            <MenuItem item={listItems}/>
        );
    });
    return(
        <ul>{updatedList}</ul>;
    )
}
const menuitems = [1, 2, 3, 4, 5];
ReactDOM.render(
    <Navmenu menuitems={menuitems}/>,
    document.getElementById('root')
);
```

The above code will run successfully without any warning message.

React Keys

Uniqueness of Keys

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

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

```
import React from 'react';
import ReactDOM from 'react-dom';
// Component to be extracted
function MenuItem(props)
{
    const item = props.item;
    return(
        <li>
            {item}
        </li>
    );
}
// Component that will return an
// unordered list
```

React Keys

```
function Navmenu(props)
{
    const list = props.menuitems;
    const updatedList = list.map((listItems)=>{
        return (
            <li>{listItems}</li>
        );
    });
    return(
        <ul>{updatedList}</ul>);
}
```

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

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

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

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} />;  
  }  
}
```

React Refs

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();
  }
}
```

React Refs

```
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;
```

React Refs

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>  
  );  
}
```

React Refs

```
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;
```

React Refs

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
```

Example:

```
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();
```

```
    };
```

```
  }
```


React Refs

```
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()`.

React Refs

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;
```

React Refs

In the previous 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 onClick = () => {  
    inputRef.current.focus();  
  };  
  return (  
    <>  
      <input ref={inputRef} type="text" />  
      <button onClick={onClick}>Submit</button>  
    </>  
  );  
}
```

Rendering Elements

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

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

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

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

```
class App extends Component {
```

```
  render() {
```

```
    return (
```

```
      <div>
```

```
        <h1>Welcome to MyInstitute!</h1>
```

```
      </div>
```

```
    );
```

```
  }
```

```
}
```

```
export default App;
```

Now, you have created your first ever React Element and also have rendered it in place, but React was not developed to create static pages, the intention of using React is to create a more logical and active webpage. In order to do so, we will need to update the elements. This next section will guide us through the same.

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

Rendering Elements

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

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

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

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

setInterval(showTime, 1000);
```

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

Rendering Elements

React Render Efficiency:

React is chosen over the legacy of DOM update because of its increased efficiency. React achieves this efficiency by using the virtual DOM and efficient differentiating algorithm. In the example of displaying the current time, at each second we call the render method, and the virtual DOM gets updated and then the differentiator checks for the particular differences in Browser DOM and the Virtual DOM and then updates only what is required such as in the given example the time is the only thing that is getting changed each time not the title “Welcome to MyInstitute!” thus React only updates the time itself making it much more efficient than conventional DOM manipulation.

Important Points to Note:

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

Conditional Rendering

There may arise a situation when we want to render something based on some condition. For example, consider the situation of handling a login/logout button. Both the buttons have different functions so they will be separate components. Now, the task is if a user is logged in then we will have to render the Logout component to display the logout button and if the user is not logged in then we will have to render the Login component to display the login button. This is what we call Conditional Rendering in ReactJS. That is to create multiple components and render them based on some conditions. This is also a kind of encapsulation supported by React.

Let us now create a page in React which will have a Message and a Button. The button will read “Login” if the user is not logged in and “Logout” if the user is logged in. We will also add some functionality to this button as upon clicking “Login” the message will read “Welcome User” and the button will change to “Logout” and upon clicking “Logout” the message will change to “Please Login” and the button will change to “Login”.

To do this, we will create a parent component named “Homepage”, two components named “Login” and “Logout” and one more component named “Message”. We will use a state variable named “isLoggedIn” to store the information about whether the user is logged in or not. The value of this variable will change according to the click of the button by the user. The Homepage component will render the Message component to display the message and it will also render one of the components among Login and Logout based on the value stored in isLoggedIn. The Message component will also return different messages based on the value of state isLoggedIn.

Let us now look at the complete program to do the above task:

Index.js:

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

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

Conditional Rendering

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

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


Conditional Rendering

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

  constructor(props)
  {
    super(props);

    this.state = {isLoggedIn : false};

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

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

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

Conditional Rendering

```
render(){  
    return(  
        <div>  
            <Message isLoggedIn = {this.state.isLoggedIn}/>  
            {  
                (this.state.isLoggedIn)?(  
                    <Logout clickFunc = {this.ifLogoutClicked} />  
                ) : (  
                    <Login clickFunc = {this.ifLoginClicked} />  
                )  
            }  
        </div>  
    );  
}  
  
ReactDOM.render(  
    <Homepage />,  
    document.getElementById('root')  
);
```

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

Conditional Rendering

Using conditions with logical && operator

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

```
{  
  condition &&  
  // This section will contain elements you want to return that will be a part of output  
}
```

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

src index.js:

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

Conditional Rendering

Preventing Component from Rendering

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

Index.js:

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

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

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

You can clearly see in the above output that the Example component is rendered twice but the <h1> element is rendered only once as on the second render of the Example component, null is returned as its rendering output.

React 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 MyInstitute. </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.

```
<React.Fragment>
  <h2> child1 </h2>
  <p> child2 </p>
  .. .....
</React.Fragment>
```

React Fragments

Example:

```
// Rendering with fragments tag
class App extends React.Component {
  render() {
    return (
      <React.Fragment>
        <h2> Hello World! </h2>
        <p> Welcome to the MyInstitute. </p>
      </React.Fragment>
    );
  }
}
```

Why we use Fragments?

The main reason to use Fragments tag is:

- It makes the execution of code faster as compared to the div tag.
- 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 MyInstitute </p>
      </>
    );
  }
}
```

React Fragments

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 Routers

React Router is a standard library for routing in React. It enables the navigation among views of various components in a React Application, allows changing the browser URL, and keeps the UI in sync with the URL.

Let us create a simple application to React to understand how the React Router works. The application will contain three components: home component, about a component, and contact component. We will use React Router to navigate between these components.

`yarn create-react-app my-router-app`

Installing React Router: React Router can be installed via npm in your React application.

`yarn add react-router-dom --save`

After installing react-router-dom, add its components to your React application.

Adding React Router Components: The main Components of React Router are:

- **BrowserRouter:** BrowserRouter is a router implementation that uses the HTML5 history API(pushState, replaceState and the popstate event) to keep your UI in sync with the URL. It is the parent component that is used to store all of the other components.
- **Route:** Route is the conditionally shown component that renders some UI when its path matches the current URL.
- **Link:** Link component is used to create links to different routes and implement navigation around the application. It works like HTML anchor tag.
- **Switch:** Switch component is used to render only the first route that matches the location rather than rendering all matching routes. Although there is no defying functionality of SWITCH tag in our application because none of the LINK paths are ever going to coincide. But let's say we have a route (Note that there is no EXACT in here), then all the Route tags are going to be processed which start with '/' (all Routes start with /). This is where we need SWITCH statement to process only one of the statements.

To add React Router components in your application, open your project directory in the editor you use and go to **app.js** file. Now, add the below given code in app.js.

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

Note: BrowserRouter is aliased as Router.

React Routers

Using React Router: To use React Router, let us first create few components in the react application. In your project directory, create a folder named **component** inside the src folder and now add 3 files named **home.js**, **about.js** and **contact.js** to the component folder.

Home.js

```
import React from 'react';
```

```
function Home () {  
    return <h1>Welcome to the world of mytutorialss!</h1>  
}
```

```
export default Home;
```

About.js

```
import React from 'react';
```

```
function About () {  
    return <div>  
        <h2>mytutorial is a computer science portal for mytutorialss!</h2>  
  
        Read more about us at :  
        <a href="https://www.mytutorialss.org/about/">  
            https://www.mytutorialss.org/about/  
        </a>  
    </div>  
}  
export default About;
```

React Routers

Contact.js

```
import React from 'react';
```

```
function Contact () {  
  return <address>
```

```
    You can find us here:<br />  
    mytutorials<br />  
    5th & 6th Floor, Royal Kapsons, A- 118, <br />  
    Sector- 136, Noida, Uttar Pradesh (201305)
```

```
  </address>
```

```
}
```

```
export default Contact;
```

Now, let us include React Router components to the application:

BrowserRouter: Add BrowserRouter aliased as Router to your app.js file in order to wrap all the other components. BrowserRouter is a parent component and can have only single child.

```
class App extends Component {  
  render() {
```

```
    return (  
      <Router>
```

```
        <div className="App">  
          </div>
```

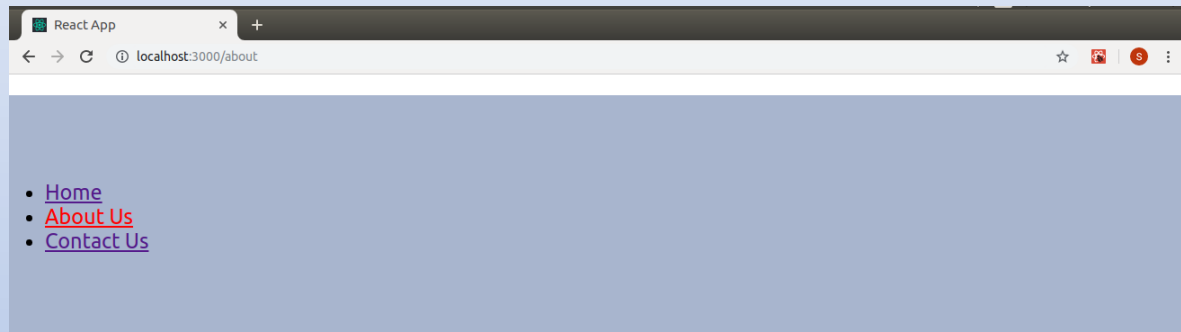
```
      </Router>
```

```
    );  
  }  
}
```

React Routers

Link: Let us now create links to our components. Link component uses the **to** prop to describe the location where the links should navigate to.

```
<div className="App">
  <ul>
    <li>
      <Link to="/">Home</Link>
    </li>
    <li>
      <Link to="/about">About Us</Link>
    </li>
    <li>
      <Link to="/contact">Contact Us</Link>
    </li>
  </ul>
</div>
```



Now, run your application on the local host and click on the links you created. You will notice the url changing according the value in **to** props of the Link component.

Route: Route component will now help us to establish the link between component's UI and the URL. To include routes to the application, add the code give below to your app.js.

```
<Route exact path="/" component={Home}></Route>
<Route exact path="/about" component={About}></Route>
<Route exact path="/contact" component={Contact}></Route>
```

Components are linked now and clicking on any link will render the component associated with it.

React Routers

Let us now try to understand the props associated with the Route component.

1. exact: It is used to match the exact value with the URL. For Eg., exact path='/about' will only render the component if it exactly matches the path but if we remove exact from the syntax, then UI will still be rendered even if the structure is like /about/10.

2. path: Path specifies a pathname we assign to our component.

3. component: It refers to the component which will render on matching the path.

Note: By default, routes are inclusive which means more than one Route component can match the URL path and render at the same time. If we want to render a single component, we need to use **switch**.

Switch: To render a single component, wrap all the routes inside the Switch Component

```
<Switch>
```

```
  <Route exact path="/" component={Home}></Route>
```

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

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

```
</Switch>
```

Switch groups together several routes, iterates over them and finds the first one that matches the path. Thereby, the corresponding component to the path is rendered.

After adding all the components here is our complete source code:

React Routers

```
import React, { Component } from 'react';
import { BrowserRouter as Router, Route, Link, Switch } from 'react-router-dom';
import Home from './component/home';
import About from './component/about';
import Contact from './component/contact';
import './App.css';
class App extends Component {
  render() {
    return (
      <Router>
        <div className="App">
          <ul className="App-header">
            <li>
              <Link to="/">Home</Link>
            </li>
            <li>
              <Link to="/about">About Us</Link>
            </li>
            <li>
              <Link to="/contact">Contact Us</Link>
            </li>
          </ul>
          <Switch>
            <Route exact path="/" component={Home}></Route>
            <Route exact path="/about" component={About}></Route>
            <Route exact path="/contact" component={Contact}></Route>
          </Switch>
        </div>
      </Router>
    );
  }
}
```

Types of Routers

On the basis of the part of URL that the router will use to track the content that the user is trying to view, React Router provides three different kinds of routers:

Memory Router

Browser Router

Hash Router

Pre-requisite: Before start this article you need to have basic knowledge of [React Router](#).

Memory Router: Memory router keeps the URL changes in memory not in the user browsers. It keeps the history of the URL in memory (does not read or write to the address bar so the user can not use the browser's back button as well as the forward button. It doesn't change the URL in your browser. It is very useful for testing and non-browser environments like React Native.

Syntax: `import { MemoryRouter as Router } from 'react-router-dom';`

Example:

```
import React, { Component } from 'react';
import { MemoryRouter as Router, Route, Link, Switch } from 'react-router-dom';
import Home from './component/home';
import About from './component/about';
import Contact from './component/contact';
import './App.css';
class App extends Component {
  render() {
    return (
      <Router>
        <div className="App">
          <ul className="App-header">
            <li>
              <Link to="/">Home</Link>
            </li>
          </ul>
        </div>
      </Router>
    );
  }
}
```

Types of Routers

```
<li>
    <Link to="/about">
        About Us
    </Link>
</li>
<li>
    <Link to="/contact">
        Contact Us
    </Link>
</li>
</ul>

<Switch>
  <Route exact path="/"
    component={Home}>
  </Route>
  <Route exact path="/about"
    component={About}>
  </Route>
  <Route exact path="/contact"
    component={Contact}>
  </Route>
</Switch>
</div>
</Router>
);
}
```

export default App;

Types of Routers

Browser Router: It uses HTML 5 history API (i.e. pushState, replaceState and popState API) to keep your UI in sync with the URL. It routes as a normal URL in the browser and assumes that the server is handling all the request URL (eg., /, /about) and points to root index.html. It accepts forceRefresh props to support legacy browsers which doesn't support HTML 5 pushState API

Syntax: `import { BrowserRouter as Router } from 'react-router-dom';`

```
import React, { Component } from 'react';
import { BrowserRouter as Router, Route, Link, Switch }
                                from 'react-router-dom';

import Home from './component/home';
import About from './component/about';
import Contact from './component/contact';
import './App.css';
class App extends Component {
  render() {
    return (
      <Router>
        <div className="App">
          <ul className="App-header">
            <li>
              <Link to="/">Home</Link>
            </li>
```


Types of Routers

```
    <li>
      <Link to="/about">About Us</Link>
    </li>
    <li>
      <Link to="/contact">
        Contact Us
      </Link>
    </li>
  </ul>
  <Switch>
    <Route exact path="/"
      component={Home}>
    </Route>
    <Route exact path="/about"
      component={About}>
    </Route>
    <Route exact path="/contact"
      component={Contact}>
    </Route>
  </Switch>
</div>
</Router>
);
}
}

export default App;
```

Types of Routers

Hash Router: Hash router uses client-side hash routing. It uses the hash portion of the URL (i.e. window.location.hash) to keep your UI in sync with the URL. Hash portion of the URL won't be handled by the server, the server will always send the index.html for every request and ignore the hash value. It doesn't need any configuration in the server to handle routes. It is used to support legacy browsers which usually don't support HTML pushState API. It is very useful for the legacy browsers or you don't have a server logic to handle the client-side. This route isn't recommended to be used by the react-router-dom team.

Syntax: `import { HashRouter as Router } from 'react-router-dom';`

```
import React, { Component } from 'react';
import { HashRouter as Router, Route, Link, Switch }
    from 'react-router-dom';

import Home from './component/home';
import About from './component/about';
import Contact from './component/contact';
import './App.css';

class App extends Component {
  render() {
    return (
      <Router>
        <div className="App">
          <ul className="App-header">
            <li>
              <Link to="/">Home</Link>
            </li>
          </ul>
        </div>
      </Router>
    );
  }
}
```

Types of Routers

```
<li>
  <Link to="/about">About Us</Link>
</li>
<li>
  <Link to="/contact">
    Contact Us
  </Link>
</li>
</ul>
<Switch>
  <Route exact path="/"
    component={Home}>
  </Route>
  <Route exact path="/about"
    component={About}>
  </Route>
  <Route exact path="/contact"
    component={Contact}>
  </Route>
</Switch>
</div>
</Router>
);
}
}

export default App;
```

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 mytutorialspoint!</h1>
        <p>Here, you can find all CS tutorials.</p>
      </div>
    );
  }
}
export default App;
```

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 mytutorials!</h1>
        <p>Here, you can find all CS tutorials.</p>
      </div>
    );
  }
}
export default App;
```

React CSS

Note: You can see in the above example, we have used two curly braces in:

```
<h1 style={{color: "Green"}}>Hello mytutorials!</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 `{{}}`.

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 mytutorials!</h1>  
        <p style={{backgroundColor: "lightgreen"}}>Here, you can find all CS tutorials.</p>  
      </div>  
    );  
  }  
}  
export default App;
```

React CSS

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 mytutorials</h1>
```

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

```
      </div>
```

```
    );
```

```
  }
```

```
}
```

```
export default App;
```

React CSS

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 mytutorials</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;
}
```


React CSS

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 mytutorials</h1>  
        <p className={styles.parastyle}>It provides great CS tutorials.</p>  
      </div>  
    );  
  }  
}  
export default App;
```

React CSS

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;
```

React CSS

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:

```
$ 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';
```

React CSS

```
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 mytutorials!!</Paragraph>
        </Div>
      </div>
    );
  }
}
```