**Introdution**

React is JS library for creating user interfaces, by facebook and Instagram

It is not another MVC framework , it is just a library for rendering your views, it just V in MVC.

**Why React?**

Simple

Declarative

Reusable

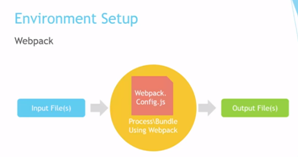
**Environment setup**

Text Editor(Atom,Bbrackets,VisualStudio,Notepad)

( react..js,react-dom.js,babel,node and web pack for package structure)

-> Npm is Node package Manager, after installing check **npm version** at command prompt , it is used to install various JS libraries and frameworks and this tool is used to install webpack, babel and react.

->Webpack



Webpack is bundling tool, take set of input files, processes them and bundle set and give us output files.

Webpackconfig file rules defined, rules are what is input file, from where we picked up, what is output file and where it should be saved.

Install webpack using

C:\Users\Aruna>npm –installwebpack –g

Then it will install modules in the following folder

C:\Users\Aruna\node\_modules

**First React Application**

**Props**

**Styles**

**States**

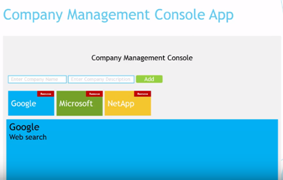
**Events**

**Refs**

**Routing**

**Lifecycle Methods**

**Building Example Application**

****

**ReactJS**

React is a front-end library developed by Facebook. It is used for handling the view layer for web and mobile apps. ReactJS allows us to create reusable UI components. It is currently one of the most popular JavaScript libraries and has a strong foundation and large community behind it.

ReactJS is JavaScript library used for building reusable UI components

React is a library for building composable user interfaces. It encourages the creation of reusable UI components, React is used as V in MVC.

React enables simpler programming model and better performance.

**Prerequisites**

JavaScript, HTML5, and CSS

**React features:**

* **JSX** − JSX is JavaScript syntax extension. It isn't necessary to use JSX in React development, but it is recommended.
* **Components** − React is all about components. In React everythingis a component. This will help you maintain the code when working on larger scale projects.
* **Unidirectional data flow and Flux** − React implements one-way data flow which makes it easy to reason about your app. Flux is a pattern that helps keeping your data unidirectional.
* **License** − React is licensed under the Facebook Inc. Documentation is licensed under CC BY 4.0.

**Advantages:**

* Uses virtual DOM which is a JavaScript object. This will improve apps performance, since JavaScript virtual DOM is faster than the regular DOM.
* Can be used on client and server side as well as with other frameworks.
* Component and data patterns improve readability, which helps to maintain larger apps.

**Limitations:**

* Covers only the view layer of the app, hence you still need to choose other technologies to get a complete tooling set for development.
* Uses inline templating and JSX, which may be difficult to handle by developers.

Angular is one of the popular frameworks used for UI development. Let us compare React and Angular in brief:

|  |  |
| --- | --- |
| **React** | **Angular** |
| React is a small view library | Angular is a full framework |
| React covers only the rendering and event handling part | Angular provides the complete solution for front-end development |
| Presentation code in JavaScript powered by JSX | Presentation code in HTML embedded with JavaScript expressions |
| React's core size is smaller than Angular, so bit fast | Angular being a framework contains a lot of code, resulting in longer load time |
| React is very flexible | Angular has less flexibility |
| Great performer, since it uses Virtual DOM | Angular uses actual DOM which affects its performance |

**JSX:**

React uses JSX for templating instead of regular JavaScript. It is not necessary to use it, however, following are some pros that come with it.

* It is faster because it performs optimization while compiling code to JavaScript.
* It is also type-safe and most of the errors can be caught during compilation.
* It makes it easier and faster to write templates, if you are familiar with HTML.

## Using JSX

JSX looks like a regular HTML in most cases. We already used it in the Environment Setup chapter. Look at the code from **App.jsx** where we are returning **div**.

### **App.jsx**

importReactfrom'react';

classAppextendsReact.Component{

render(){

return(

<div>

HelloWorld!!!

</div>

);

}

}

exportdefaultApp;

Even though it's similar to HTML, there are a couple of things we need to keep in mind when working with JSX.

## Nested Elements

If we want to return more elements, we need to wrap it with one container element. Notice how we are using **div** as a wrapper for **h1**, **h2** and **p**elements.

### **App.jsx**

importReactfrom'react';

classAppextendsReact.Component{

render(){

return(

<div>

<h1>Header</h1>

<h2>Content</h2>

<p>Thisis the content!!!</p>

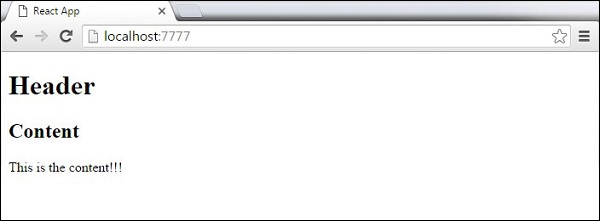
</div>

);

}

}

exportdefaultApp;



## Attributes

We can use our own custom attributes in addition to regular HTML properties and attributes. When we want to add custom attribute, we need to use **data-**prefix. In the following example, we added **data-myattribute** as an attribute of **p** element.

importReactfrom'react';

classAppextendsReact.Component{

render(){

return(

<div>

<h1>Header</h1>

<h2>Content</h2>

<p data-myattribute="somevalue">Thisis the content!!!</p>

</div>

);

}

}

exportdefaultApp;

## JavaScript Expressions

JavaScript expressions can be used inside of JSX. We just need to wrap it with curly brackets **{}**. The following example will render **2**.

importReactfrom'react';

classAppextendsReact.Component{

render(){

return(

<div>

<h1>{1+1}</h1>

</div>

);

}

}

exportdefaultApp;



We cannot use **if else** statements inside JSX, instead we can use **conditional (ternary)** expressions. In the following example, variable **i** equals to **1** so the browser will render **true**, If we change it to some other value, it will render **false**.

importReactfrom'react';

classAppextendsReact.Component{

render(){

vari=1;

return(

<div>

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

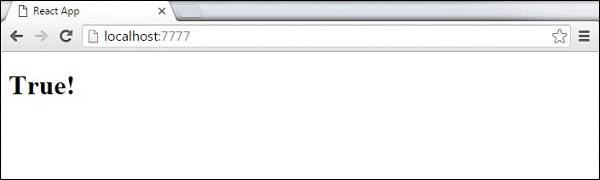
</div>

);

}

}

exportdefaultApp;



## Styling

React recommends using inline styles. When we want to set inline styles, we need to use **camelCase** syntax. React will also automatically append **px** after the number value on specific elements. The following example shows how to add **myStyle** inline to **h1** element.

importReactfrom'react';

classAppextendsReact.Component{

render(){

varmyStyle={

fontSize:100,

color:'#FF0000'

}

return(

<div>

<h1 style ={myStyle}>Header</h1>

</div>

);

}

}

exportdefaultApp;



## Comments

When writing comments, we need to put curly brackets **{}** when we want to write comment within children section of a tag. It is a good practice to always use **{}** when writing comments, since we want to be consistent when writing the app.

importReactfrom'react';

classAppextendsReact.Component{

render(){

return(

<div>

<h1>Header</h1>

{//End of the line Comment...}

{/\*Multi line comment...\*/}

</div>

);

}

}

exportdefaultApp;

## Naming Convention

HTML tags always use **lowercase** tag names, while React components start with **Uppercase**.

**Note** − You should use **className** and **htmlFor** as XML attribute names instead of **class** and **for**.

This is explained on React official page as −

Since JSX is JavaScript, identifiers such as **class** and **for** are discouraged as XML attribute names. Instead, React DOM components expect DOM property names such as **className** and **htmlFor**, respectively.

**ReactJS Components:**

React JS components can be combined to make the app easier to maintain. This approach allows to update and change your components without affecting the rest of the page.

## Stateless Example

Our first component in the following example is **App**. This component is owner of **Header** and **Content**. We are creating **Header** and **Content** separately and just adding it inside JSX tree in our **App** component. Only **App** component needs to be exported.

### **App.jsx**

importReactfrom'react';

classAppextendsReact.Component{

render(){

return(

<div>

<Header/>

<Content/>

</div>

);

}

}

classHeaderextendsReact.Component{

render(){

return(

<div>

<h1>Header</h1>

</div>

);

}

}

classContentextendsReact.Component{

render(){

return(

<div>

<h2>Content</h2>

<p>The content text!!!</p>

</div>

);

}

}

exportdefaultApp;

To be able to render this on the page, we need to import it in **main.js** file and call **reactDOM.render()**. We already did this while setting the environment.

### **main.js**

importReactfrom'react';

importReactDOMfrom'react-dom';

importAppfrom'./App.jsx';

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

The above code will generate the following result.



## Stateful Example

In this example, we will set the state for owner component (**App**). The **Header** component is just added like in the last example since it doesn't need any state. Instead of content tag, we are creating **table** and **tbody** elements, where we will dynamically insert **TableRow** for every object from the **data**array.

It can be seen that we are using EcmaScript 2015 arrow syntax (**⇒**) which looks much cleaner than the old JavaScript syntax. This will help us create our elements with fewer lines of code. It is especially useful when we need to create a list with a lot of items.

### **App.jsx**

importReactfrom'react';

classAppextendsReact.Component{

constructor(){

super();

this.state={

data:

[

{

"id":1,

"name":"Foo",

"age":"20"

},

{

"id":2,

"name":"Bar",

"age":"30"

},

{

"id":3,

"name":"Baz",

"age":"40"

}

]

}

}

render(){

return(

<div>

<Header/>

<table>

<tbody>

{this.state.data.map((person,i)=><TableRow key ={i}

data ={person}/>)}

</tbody>

</table>

</div>

);

}

}

classHeaderextendsReact.Component{

render(){

return(

<div>

<h1>Header</h1>

</div>

);

}

}

classTableRowextendsReact.Component{

render(){

return(

<tr>

<td>{this.props.data.id}</td>

<td>{this.props.data.name}</td>

<td>{this.props.data.age}</td>

</tr>

);

}

}

exportdefaultApp;

### **main.js**

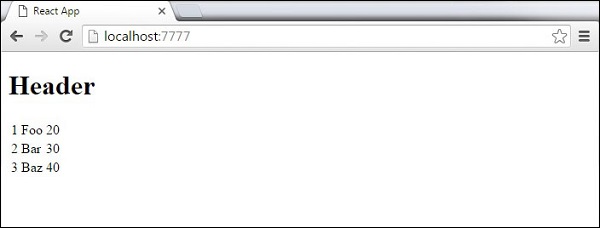
importReactfrom'react';

importReactDOMfrom'react-dom';

importAppfrom'./App.jsx';

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

**Note** − Notice that we are using **key = {i} inside map()** function. This will help React to update only the necessary elements instead of re-rendering the entire list when something changes. It is a huge performance boost for larger number of dynamically created elements.



**ReactJS state:**

**State** is the place where the data comes from. We should always try to make our state as simple as possible and minimize the number of stateful components. If we have, for example, ten components that need data from the state, we should create one container component that will keep the state for all of them.

## Using Props

The following sample code shows how to create a stateful component using EcmaScript2016 syntax.

### **App.jsx**

importReactfrom'react';

classAppextendsReact.Component{

constructor(props){

super(props);

this.state={

header:"Header from state...",

content:"Content from state..."

}

}

render(){

return(

<div>

<h1>{this.state.header}</h1>

<h2>{this.state.content}</h2>

</div>

);

}

}

exportdefaultApp;

### **main.js**

importReactfrom'react';

importReactDOMfrom'react-dom';

importAppfrom'./App.jsx';

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

This will produce the following result.



**Properties validation**

Properties validation is a useful way to force the correct usage of the components. This will help during development to avoid future bugs and problems, once the app becomes larger. It also makes the code more readable, since we can see how each component should be used.

## Validating Props

In this example, we are creating **App** component with all the **props** that we need. **App.propTypes** is used for props validation. If some of the props aren't using the correct type that we assigned, we will get a console warning. After we specify validation patterns, we will set **App.defaultProps**.

### **App.jsx**

importPropTypesfrom'prop-types';

importReactfrom'react';

importReactDOMfrom'react-dom';

classAppextendsReact.Component{

render(){

return(

<div>

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

<h3>Array:{this.props.propArray}</h3>

<h3>Bool:{this.props.propBool?"True...":"False..."}</h3>

<h3>Func:{this.props.propFunc(3)}</h3>

<h3>Number:{this.props.propNumber}</h3>

<h3>String:{this.props.propString}</h3>

</div>

);

}

}

App.propTypes={

name:PropTypes.string,

propArray:PropTypes.array.isRequired,

propBool:PropTypes.bool.isRequired,

propFunc:PropTypes.func,

propNumber:PropTypes.number,

propString:PropTypes.string,

};

App.defaultProps={

name:'Tutorialspoint.com',

propArray:[1,2,3,4,5],

propBool:true,

propFunc:function(e){

return e

},

propNumber:1,

propString:"String value..."

}

exportdefaultApp;

### **main.js**

importReactfrom'react';

importPropTypesfrom'prop-types';

importReactDOMfrom'react-dom';

importAppfrom'./App.jsx';

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

### **webpack.config.js**

varconfig={

entry:'./main.js',

output:{

path:'/',

filename:'index.js',

},

devServer:{

inline:true,

port:8080

},

externals:{

'react':'React'

},

module:{

loaders:[

{

test:/\.jsx?$/,

exclude:/node\_modules/,

loader:'babel-loader',

query:{

presets:['es2015','react']

}

}

]

}

}

module.exports=config;

Since all **props** are valid, we will get the following result.

As can be noticed, we have use **isRequired** when validating **propArray** and **propBool**. This will give us an error, if one of those two don't exist. If we delete **propArray:** **[1,2,3,4,5]** from the **App.defaultProps** object, the console will log a warning.

React Props Validation Error

If we set the value of **propArray: 1**, React will warn us that the propType validation has failed, since we need an array and we got a number.

React Props Validation Error 2

**React component API.**

We will discuss three methods: **setState(), forceUpdate** and **ReactDOM.findDOMNode()**. In new ES6 classes, we have to manually bind this. We will use **this.method.bind(this)** in the examples.

Set State

**setState()** method is used to update the state of the component. This method will not replace the state, but only add changes to the original state.

importReactfrom'react';

classAppextendsReact.Component{

constructor(){

super();

this.state={

data:[]

}

this.setStateHandler=this.setStateHandler.bind(this);

};

setStateHandler(){

var item ="setState..."

varmyArray=this.state.data.slice();

myArray.push(item);

this.setState({data:myArray})

};

render(){

return(

<div>

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

<h4>StateArray:{this.state.data}</h4>

</div>

);

}

}

exportdefaultApp;

We started with an empty array. Every time we click the button, the state will be updated. If we click five times, we will get the following output.



Force Update

Sometimes we might want to update the component manually. This can be achieved using the **forceUpdate()** method.

importReactfrom'react';

classAppextendsReact.Component{

constructor(){

super();

this.forceUpdateHandler=this.forceUpdateHandler.bind(this);

};

forceUpdateHandler(){

this.forceUpdate();

};

render(){

return(

<div>

<button onClick={this.forceUpdateHandler}>FORCE UPDATE</button>

<h4>Random number:{Math.random()}</h4>

</div>

);

}

}

exportdefaultApp;

We are setting a random number that will be updated every time the button is clicked.



Find Dom Node

For DOM manipulation, we can use **ReactDOM.findDOMNode()** method. First we need to import **react-dom**.

importReactfrom'react';

importReactDOMfrom'react-dom';

classAppextendsReact.Component{

constructor(){

super();

this.findDomNodeHandler=this.findDomNodeHandler.bind(this);

};

findDomNodeHandler(){

varmyDiv=document.getElementById('myDiv');

ReactDOM.findDOMNode(myDiv).style.color='green';

}

render(){

return(

<div>

<button onClick={this.findDomNodeHandler}>FIND DOME NODE</button>

<div id ="myDiv">NODE</div>

</div>

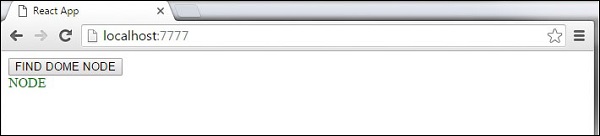
);

}

}

exportdefaultApp;

The color of **myDiv** element changes to green, once the button is clicked.



**Note** − Since the 0.14 update, most of the older component API methods are deprecated or removed to accommodate ES6.

## Lifecycle Methods

* **componentWillMount** is executed before rendering, on both the server and the client side.
* **componentDidMount** is executed after the first render only on the client side. This is where AJAX requests and DOM or state updates should occur. This method is also used for integration with other JavaScript frameworks and any functions with delayed execution such as **setTimeout** or **setInterval**. We are using it to update the state so we can trigger the other lifecycle methods.
* **componentWillReceiveProps** is invoked as soon as the props are updated before another render is called. We triggered it from **setNewNumber** when we updated the state.
* **shouldComponentUpdate** should return **true** or **false** value. This will determine if the component will be updated or not. This is set to **true** by default. If you are sure that the component doesn't need to render after **state** or **props** are updated, you can return **false** value.
* **componentWillUpdate** is called just before rendering.
* **componentDidUpdate** is called just after rendering.
* **componentWillUnmount** is called after the component is unmounted from the dom. We are unmounting our component in **main.js**.

In the following example, we will set the initial **state** in the constructor function. The **setNewnumber** is used to update the **state**. All the lifecycle methods are inside the Content component.

### **App.jsx**

importReactfrom'react';

classAppextendsReact.Component{

constructor(props){

super(props);

this.state={

data:0

}

this.setNewNumber=this.setNewNumber.bind(this)

};

setNewNumber(){

this.setState({data:this.state.data+1})

}

render(){

return(

<div>

<button onClick={this.setNewNumber}>INCREMENT</button>

<ContentmyNumber={this.state.data}></Content>

</div>

);

}

}

classContentextendsReact.Component{

componentWillMount(){

console.log('Component WILL MOUNT!')

}

componentDidMount(){

console.log('Component DID MOUNT!')

}

componentWillReceiveProps(newProps){

console.log('Component WILL RECIEVE PROPS!')

}

shouldComponentUpdate(newProps,newState){

returntrue;

}

componentWillUpdate(nextProps,nextState){

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

}

componentDidUpdate(prevProps,prevState){

console.log('Component DID UPDATE!')

}

componentWillUnmount(){

console.log('Component WILL UNMOUNT!')

}

render(){

return(

<div>

<h3>{this.props.myNumber}</h3>

</div>

);

}

}

exportdefaultApp;

### **main.js**

importReactfrom'react';

importReactDOMfrom'react-dom';

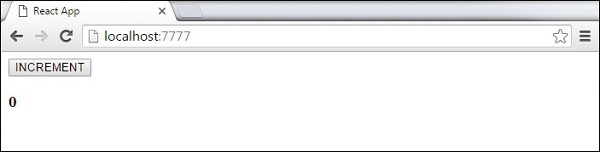
importAppfrom'./App.jsx';

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

setTimeout(()=>{

ReactDOM.unmountComponentAtNode(document.getElementById('app'));},10000);

After the initial render, we will get the following screen.



Only **componentWillMount** and **componentDidMount** will be logged in the console, since we didn't update anything yet.

React Component Lifecycle Initial Log

When we click the **INCREMENT** button, the update will occur and other lifecycle methods will be triggered.

React Component Lifecycle Change Log

After ten seconds, the component will unmount and the last event will be logged in the console.

React Component Lifecycle Unmount Log

**Note** − Lifecycle methods will always be invoked in the same order so it is a good practice to write it in the correct order as shown in the example.

**ReactJS forms:**

In the following example, we will set an input form with **value = {this.state.data}**. This allows to update the state whenever the input value changes. We are using **onChange** event that will watch the input changes and update the state accordingly.

### **App.jsx**

importReactfrom'react';

classAppextendsReact.Component{

constructor(props){

super(props);

this.state={

data:'Initial data...'

}

this.updateState=this.updateState.bind(this);

};

updateState(e){

this.setState({data:e.target.value});

}

render(){

return(

<div>

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

onChange={this.updateState}/>

<h4>{this.state.data}</h4>

</div>

);

}

}

exportdefaultApp;

### **main.js**

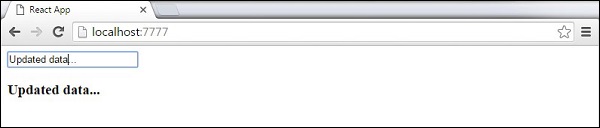
importReactfrom'react';

importReactDOMfrom'react-dom';

importAppfrom'./App.jsx';

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

When the input text value changes, the state will be updated.



## Complex Example

In the following example, we will see how to use forms from child component. **onChange** method will trigger state update that will be passed to the child input **value** and rendered on the screen. A similar example is used in the Events chapter. Whenever we need to update state from child component, we need to pass the function that will handle updating (**updateState**) as a prop (**updateStateProp**).

### **App.jsx**

importReactfrom'react';

classAppextendsReact.Component{

constructor(props){

super(props);

this.state={

data:'Initial data...'

}

this.updateState=this.updateState.bind(this);

};

updateState(e){

this.setState({data:e.target.value});

}

render(){

return(

<div>

<ContentmyDataProp={this.state.data}

updateStateProp={this.updateState}></Content>

</div>

);

}

}

classContentextendsReact.Component{

render(){

return(

<div>

<input type ="text" value ={this.props.myDataProp}

onChange={this.props.updateStateProp}/>

<h3>{this.props.myDataProp}</h3>

</div>

);

}

}

exportdefaultApp;

### **main.js**

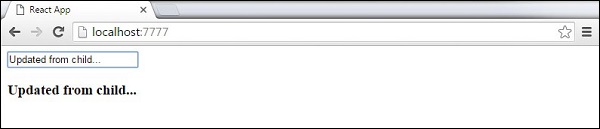
importReactfrom'react';

importReactDOMfrom'react-dom';

importAppfrom'./App.jsx';

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

This will produce the following result.



This is a simple example where we will only use one component. We are just adding **onClick** event that will trigger **updateState** function once the button is clicked.

### **App.jsx**

importReactfrom'react';

classAppextendsReact.Component{

constructor(props){

super(props);

this.state={

data:'Initial data...'

}

this.updateState=this.updateState.bind(this);

};

updateState(){

this.setState({data:'Data updated...'})

}

render(){

return(

<div>

<button onClick={this.updateState}>CLICK</button>

<h4>{this.state.data}</h4>

</div>

);

}

}

exportdefaultApp;

### **main.js**

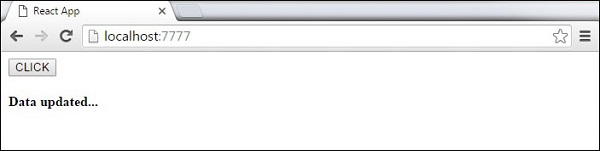
importReactfrom'react';

importReactDOMfrom'react-dom';

importAppfrom'./App.jsx';

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

This will produce the following result.



## Child Events

When we need to update the **state** of the parent component from its child, we can create an event handler (**updateState**) in the parent component and pass it as a prop (**updateStateProp**) to the child component where we can just call it.

### **App.jsx**

importReactfrom'react';

classAppextendsReact.Component{

constructor(props){

super(props);

this.state={

data:'Initial data...'

}

this.updateState=this.updateState.bind(this);

};

updateState(){

this.setState({data:'Data updated from the child component...'})

}

render(){

return(

<div>

<ContentmyDataProp={this.state.data}

updateStateProp={this.updateState}></Content>

</div>

);

}

}

classContentextendsReact.Component{

render(){

return(

<div>

<button onClick={this.props.updateStateProp}>CLICK</button>

<h3>{this.props.myDataProp}</h3>

</div>

);

}

}

exportdefaultApp;

### **main.js**

importReactfrom'react';

importReactDOMfrom'react-dom';

importAppfrom'./App.jsx';

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

This will produce the following result.



**ReactJs refs:**

The **ref** is used to return a reference to the element. **Refs** should be avoided in most cases, however, they can be useful when we need DOM measurements or to add methods to the components.

## Using Refs

The following example shows how to use refs to clear the input field. **ClearInput** function searches for element with **ref = "myInput"** value, resets the state, and adds focus to it after the button is clicked.

### **App.jsx**

importReactfrom'react';

importReactDOMfrom'react-dom';

classAppextendsReact.Component{

constructor(props){

super(props);

this.state={

data:''

}

this.updateState=this.updateState.bind(this);

this.clearInput=this.clearInput.bind(this);

};

updateState(e){

this.setState({data:e.target.value});

}

clearInput(){

this.setState({data:''});

ReactDOM.findDOMNode(this.refs.myInput).focus();

}

render(){

return(

<div>

<input value ={this.state.data}onChange={this.updateState}

ref="myInput"></input>

<button onClick={this.clearInput}>CLEAR</button>

<h4>{this.state.data}</h4>

</div>

);

}

}

exportdefaultApp;

### **main.js**

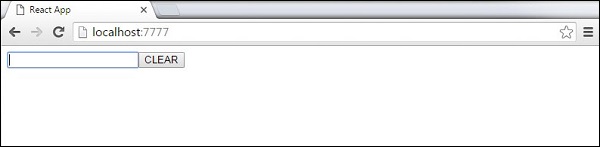
importReactfrom'react';

importReactDOMfrom'react-dom';

importAppfrom'./App.jsx';

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

Once the button is clicked, the **input** will be cleared and focused.



**ReactJS keys:**

React **keys** are useful when working with dynamically created components or when your lists are altered by the users. Setting the **key** value will keep your components uniquely identified after the change.

## Using Keys

Let's dynamically create **Content** elements with unique index (i). The **map**function will create three elements from our **data** array. Since the **key** value needs to be unique for every element, we will assign i as a key for each created element.

### **App.jsx**

importReactfrom'react';

classAppextendsReact.Component{

constructor(){

super();

this.state={

data:[

{

component:'First...',

id:1

},

{

component:'Second...',

id:2

},

{

component:'Third...',

id:3

}

]

}

}

render(){

return(

<div>

<div>

{this.state.data.map((dynamicComponent,i)=><Content

key ={i}componentData={dynamicComponent}/>)}

</div>

</div>

);

}

}

classContentextendsReact.Component{

render(){

return(

<div>

<div>{this.props.componentData.component}</div>

<div>{this.props.componentData.id}</div>

</div>

);

}

}

exportdefaultApp;

### **main.js**

importReactfrom'react';

importReactDOMfrom'react-dom';

importAppfrom'./App.jsx';

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

We will get the following result for the Key values of each element.



If we add or remove some elements in the future or change the order of the dynamically created elements, React will use the **key** values to keep track of each element.

**ReactJs Router:**

## Step 1 - Install a React Router

A simple way to install the **react-router** is to run the following code snippet in the **command prompt** window.

C:\Users\username\Desktop\reactApp>npm install react-router-dom

Configure the .babelrc file in the root of the project as shown below-

C:\Users\username\Desktop\reactApp>\type nul>.babelrc

Add the following code in **.babelrc** file

{

"presets": ["es2015", "react"]

}

Create a file as index.html in a root directory and the following code -

<!DOCTYPE html>

<htmllang="en">

<head>

<metacharset="UTF-8">

<title>React Router Tutorial</title>

</head>

<body>

<divid="app"></div>

<scripttype="text/javascript"src="bundle.js"></script>

</body>

</html>

To configure webpack.config.js file, add the following code in webpack.config.js

module.exports = {

entry: './app/main.js',

output: {

filename: 'bundle.js'

},

module: {

loaders: [

{

loader: 'babel-loader',

test: /\.js$/,

exclude: /node\_modules/

}

]

},

devServer: {

port: 7777

}

};

## Step 2 - Add a Router

Now, we will add routes to the app. Instead of rendering **App** element like in the previous example, create a directory named as **App** and create files and named as main.js and App.js

### **Main.js**

importReactfrom'react';

import{ render }from'react-dom';

importAppfrom'./App';

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

### **App.js**

importReact,{Component}from'react';

import{BrowserRouterasRouter,Switch,Route,Link}from'react-router-dom';

importHomefrom'./Home';

importLoginfrom'./Login';

classAppextendsComponent{

render(){

return(

<Router>

<div>

<h2>Welcome to ReactRouterTutorial</h2>

<ul>

<li><Link to={'/'}>Home</Link></li>

<li><Link to={'/Login'}>Login</Link></li>

</ul>

<hr/>

<Switch>

<Route exact path='/' component={Home}/>

<Route exact path='/Login' component={Login}/>

</Switch>

</div>

</Router>

);

}

}

exportdefaultApp;

## Step 3 - Create Components

In this step, we will create two components as (**Home**)and (**Login**) in App directory.

### **Home.js**

importReact,{Component}from'react';

classHomeextendsComponent{

render(){

return(

<div>

<h2>Home</h2>

</div>

);

}

}

exportdefaultHome;

### **Login.js**

importReact,{Component}from'react';

importReactDOMfrom'react-dom';

classLoginextendsComponent{

render(){

return(

<div>

<h2>Login</h2>

</div>

);

}

}

exportdefaultLogin;

When the app is started, we will see two clickable links that can be used to change the route.

