

**React** is a front-end library developed by **Facebook**. React is a library for building composable user interfaces. It encourages the creation of reusable UI components, which present data those 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.

**Evolution Of React**

React is a JavaScript library used to build the user interface for web applications. React was initially developed and maintained by the folks at Facebook, which was later used in their products (WhatsApp & Instagram). Now it is an open source project with an active developer community. Popular websites like Netflix, Airbnb, Yahoo!Mail, KhanAcademy, Drop box and many more use React to build their UI. Modern websites are built using MVC (model view controller) architecture. React is the ‘V’ in the MVC which stands for view, whereas the architecture is provided by **Redux** or **Flux**. React native is used to develop mobile apps, the Facebook mobile app is built using React native.

Facebook’s annual F8 Developer conference 2017 saw two promising announcements: **React Fiber** and **ReactVR**. React Fiber is a complete rewrite of the previous release focusing on incremental rendering and quick responsiveness, React Fiber is backward compatible with all previous versions. ReactVR is built on top of React Native frameworks, it enables developing UI with the addition of 3D models to replicate 360-degree environment resulting in fully immersive VR content.

**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. You need to think of everything as 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.

**React 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.

**React 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 might seem awkward to some developers.

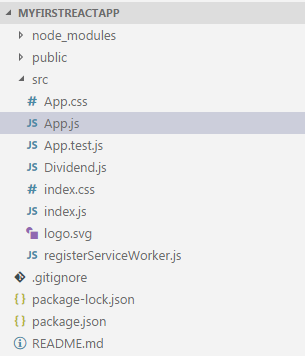
**Hello World Application**

Step 1) Install NodeJS, Visual Studio Code

Step 2) Use below command to create a sample React JS Application

**npx create-react-app myfirstreactapp**

Step 3) Import the project into the Visual Studio Code editor



Step 4) Add below code into **App.js**

import React from 'react';

class App extends React.Component {

render() {

return (

<div>

Hello World!!!

</div>

);

}

}

export default App;

The entire application can be modeled as a set of independent components. Different components are used to serve different purposes. This enables us to keep logic and views separate. React renders multiple components simultaneously. Components can be either stateful or stateless.

Before we start creating components, we need to include a few ‘import’ statements. In the first line, we have to instruct JavaScript to import the ‘react’ library from the installed ‘npm’ module. This takes care of all the dependencies needed by React.

|  |  |
| --- | --- |
|  | **import React from 'react';** |

The HTML generated by the component needs to be displayed on to the DOM, we achieve this by specifying a render function which tells React where exactly it needs to be rendered (displayed) on the screen. For this, we make a reference to an existing DOM node by passing a container element.

In React, the DOM is part of the ‘react-dom’ library. So in the next line, we have to instruct JavaScript to import ‘react-dom’ library from the installed npm module.

|  |  |
| --- | --- |
|  | **import ReactDOM from 'react-dom';** |

Step 5) Add below code into **index.js**

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App.jsx';

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

Step 6) Open Command prompt and enter below command

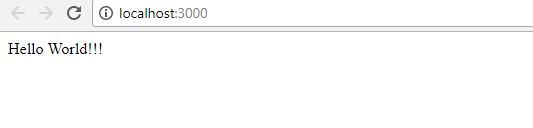
**npm install**

This will create a folder node\_modules inside your project which contains all the dependencies

Step 7) Open Command prompt and enter below command

**npm start**

This will start the server and React JS application gets executed in browser on port 3000

****

* **ReactJS - 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.js** where we are returning **div**.

### App.js

import React from 'react';

class App extends React.Component {

render() {

return (

<div>

Hello World!!!

</div>

);

}

}

export default App;

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.js

import React from 'react';

class App extends React.Component {

render() {

return (

<div>

<h1>Header</h1>

<h2>Content</h2>

<p>This is the content!!!</p>

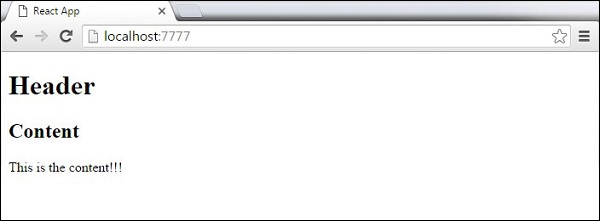
</div>

);

}

}

export default App;



## 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.

import React from 'react';

class App extends React.Component {

render() {

return (

<div>

<h1>Header</h1>

<h2>Content</h2>

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

</div>

);

}

}

export default App;

## 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**.

import React from 'react';

class App extends React.Component {

render() {

return (

<div>

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

</div>

);

}

}

export default App;



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**.

import React from 'react';

class App extends React.Component {

render() {

var i = 1;

return (

<div>

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

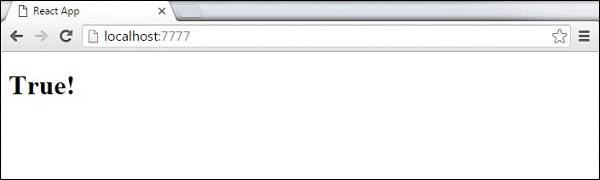
</div>

);

}

}

export default App;



## 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.

import React from 'react';

class App extends React.Component {

render() {

var myStyle = {

fontSize: 100,

color: '#FF0000'

}

return (

<div>

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

</div>

);

}

}

export default App;



## 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.

import React from 'react';

class App extends React.Component {

render() {

return (

<div>

<h1>Header</h1>

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

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

</div>

);

}

}

export default App;

* **ReactJS – Components :**

## import React from 'react';

class MyComponent extends React.Component {  
 render () {  
 return(<div> This is a component </div>);  
 }  
}

## 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

import React from 'react';

class App extends React.Component {

render() {

return (

<div>

<Header/>

<Content/>

</div>

);

}

}

class Header extends React.Component {

render() {

return (

<div>

<h1>Header</h1>

</div>

);

}

}

class Content extends React.Component {

render() {

return (

<div>

<h2>Content</h2>

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

</div>

);

}

}

export default App;

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

import React from 'react';

import ReactDOM from 'react-dom';

import App from './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.js

import React from 'react';

class App extends React.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>

);

}

}

class Header extends React.Component {

render() {

return (

<div>

<h1>Header</h1>

</div>

);

}

}

class TableRow extends React.Component {

render() {

return (

<tr>

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

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

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

</tr>

);

}

}

export default App;

### main.js

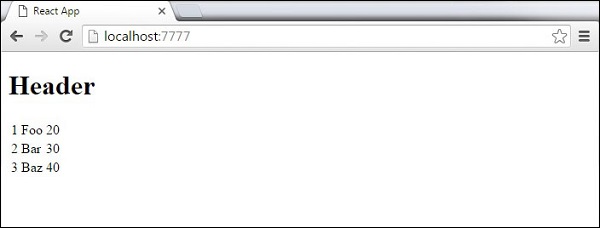
import React from 'react';

import ReactDOM from 'react-dom';

import App from './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.



**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.

import React from 'react';

class App extends React.Component {

constructor() {

super();

this.state = {

data: []

}

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

};

setStateHandler() {

var item = "setState..."

var myArray = this.state.data.slice();

myArray.push(item);

this.setState({data: myArray})

};

render() {

return (

<div>

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

<h4>State Array: {this.state.data}</h4>

</div>

);

}

}

export default App;

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.

import React from 'react';

class App extends React.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>

);

}

}

export default App;

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**.

import React from 'react';

import ReactDOM from 'react-dom';

class App extends React.Component {

constructor() {

super();

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

};

findDomNodeHandler() {

var myDiv = 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>

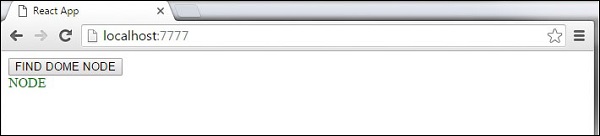
);

}

}

export default App;

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



**Component Lifecycle:**

## 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.js

import React from 'react';

class App extends React.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>

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

</div>

);

}

}

class Content extends React.Component {

componentWillMount() {

console.log('Component WILL MOUNT!')

}

componentDidMount() {

console.log('Component DID MOUNT!')

}

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!')

}

render() {

return (

<div>

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

</div>

);

}

}

export default App;

### main.js

import React from 'react';

import ReactDOM from 'react-dom';

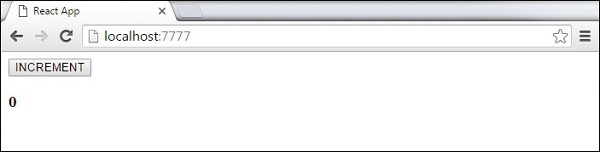
import App from './App.jsx';

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 – 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.

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

### App.js

import React from 'react';

class App extends React.Component {

constructor() {

super();

this.state = {

header: "Header from state...",

content: "Content from state..."

}

}

render() {

return (

<div>

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

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

</div>

);

}

} export default App;

### main.js

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App.jsx';

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

This will produce the following result.



* **ReactJS – Props:**

The main difference between state and props is that **props** are immutable. This is why the container component should define the state that can be updated and changed, while the child components should only pass data from the state using props.

1. When we need immutable data in our component, we can just add props to reactDOM.render() function in main.js and use it inside our component.

### App.js

import React from 'react';

class App extends React.Component {

render() {

return (

<div>

<h1>{this.props.headerProp}</h1>

<h2>{this.props.contentProp}</h2>

</div>

);

}

}

export default App;

### main.js

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App.jsx';

ReactDOM.render(<App headerProp = "Header from props..." contentProp = "Content

from props..."/>, document.getElementById('app'));

export default App;

This will produce the following result.



## Default Props

You can also set default property values directly on the component constructor instead of adding it to the **reactDom.render()** element.

### App.js

import React from 'react';

class App extends React.Component {

render() {

return (

<div>

<h1>{this.props.headerProp}</h1>

<h2>{this.props.contentProp}</h2>

</div>

);

}

}

App.defaultProps = {

headerProp: "Header from props...",

contentProp:"Content from props..."

}

export default App;

### main.js

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App.jsx';

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

Output is the same as before.



## State and Props

The following example shows how to combine state and props in your app. We are setting the state in our parent component and passing it down the component tree using props. Inside the render function, we are setting headerProp and contentProp used in child components.

**App.js**

import React from 'react';

class App extends React.Component {

constructor(props) {

super(props);

this.state = {

header: "Header from props...",

content: "Content from props..."

}

}

render() {

return (

<div>

<Header headerProp = {this.state.header}/>

<Content contentProp = {this.state.content}/>

</div>

);

}

}

class Header extends React.Component {

render() {

return (

<div>

<h1>{this.props.headerProp}</h1>

</div>

);

}

}

class Content extends React.Component {

render() {

return (

<div>

<h2>{this.props.contentProp}</h2>

</div>

);

}

}

export default App;

### main.js

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App.jsx';

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

The result will again be the same as in the previous two examples; the only thing that is different is the source of our data, which is now originally coming from the **state**. When we want to update it, we just need to update the state, and all child components will be updated.

* **ReactJS – Forms**

## Simple Example:

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.js

import React from 'react';

class App extends React.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>

);

}

}

export default App;

### main.js

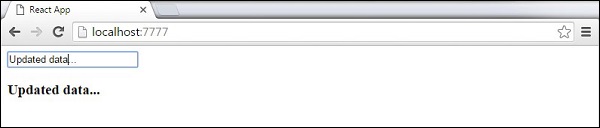
import React from 'react';

import ReactDOM from 'react-dom';

import App from './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.js

import React from 'react';

class App extends React.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>

<Content myDataProp = {this.state.data}

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

</div>

);

}

}

class Content extends React.Component {

render() {

return (

<div>

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

onChange = {this.props.updateStateProp} />

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

</div>

);

}

}

export default App;

### main.js

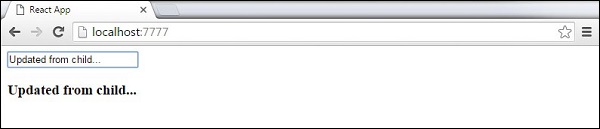
import React from 'react';

import ReactDOM from 'react-dom';

import App from './App.jsx';

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

This will produce the following result.



* **ReactJS – Events**

## Simple Example:

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.js

import React from 'react';

class App extends React.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>

);

}

}

export default App;

### main.js

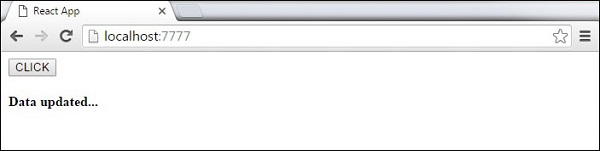
import React from 'react';

import ReactDOM from 'react-dom';

import App from './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.js

import React from 'react';

class App extends React.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>

<Content myDataProp = {this.state.data}

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

</div>

);

}

}

class Content extends React.Component {

render() {

return (

<div>

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

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

</div>

);

}

}

export default App;

### main.js

import React from 'react';

import ReactDOM from 'react-dom';

import App from './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.js

import React from 'react';

import ReactDOM from 'react-dom';

class App extends React.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>

);

}

}

export default App;

### main.js

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App.jsx';

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

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

