ReactJS - Component API

we will explain 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.

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>

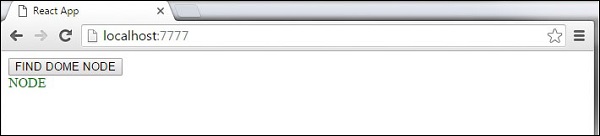
);

}

}

export default App;

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



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

# ReactJS - Component Life Cycle

## **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**

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';

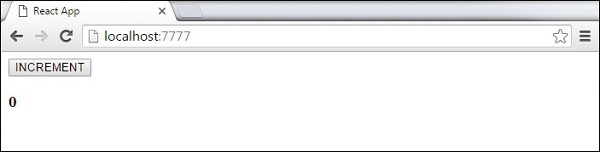
import App from './App.jsx';

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 - 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.jsx**

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**

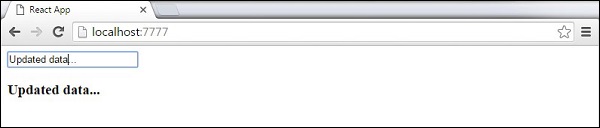
import React from 'react';

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.jsx**

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**

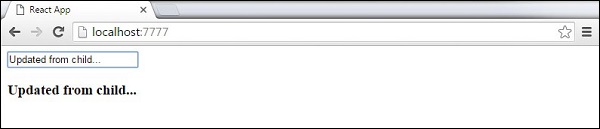
import React from 'react';

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.jsx**

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**

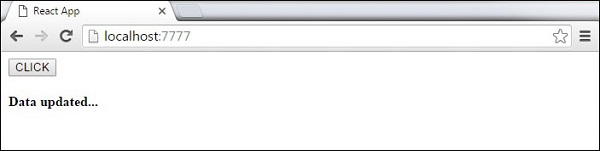
import React from 'react';

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.jsx**

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.

