# UNIT- V

**React JS:**

Why React JS, React JSX, React JS components, State, Props, React Component API and Life cycle, Forms and Events. **[10 Hours]**

## *What is React?*

**React is a JavaScript library for building user interfaces.** It is the declarative library for designing interactive web applications.At the heart of **all React applications are components**. **A component is a self-contained module that renders some output.** We can write interface elements like a button or an input field as a React component. Components are composable. A component might include one or more other components in its output.Broadly speaking, to write React apps, we write React components that correspond to various interface elements. We then organize these components inside higher-level components which define the structure of our application.

## How does it work?

Unlike many of its predecessors, React doesn't directly manipulate the browser's Document Object Model (DOM) immediately. Instead, **it operates on a virtual DOM**. This means that rather than making changes to the actual document in the browser after modifications to data (which can be slow), React resolves these changes on a DOM constructed and run entirely in memory. Once the virtual DOM is updated, React intelligently determines the necessary changes to apply to the real browser's DOM.

The **React Virtual DOM** exists solely in memory and serves as a representation of the web browser's DOM. Consequently, when writing a React component, developers aren't making direct edits to the DOM; they are editing a virtual component that React will later translate into the actual DOM. **This approach enhances performance and allows React to efficiently manage updates and changes in the user interface.**

## *What is JSX?*

JSX stands for JavaScript XML. It is a syntax extension for JavaScript that looks similar to XML or HTML. **JSX provides a concise and expressive way to describe the structure of a user interface in React applications.**

In React, instead of using regular JavaScript to define UI components, JSX allows developers to write HTML-like code directly in their JavaScript files. JSX is not a separate scripting language; it's a syntax extension that gets transformed into regular JavaScript during the build process.

Here's a simple example of JSX:

```
const element = <h1>Hello, React!</h1>;
```

In this example, the JSX syntax **<h1>Hello, React!</h1>** is used to create a React element representing an **<h1>** heading with the text **"Hello, React!".** When React processes this JSX, it translates it into JavaScript equivalent, similar to the following:

```
const element = React.createElement('h1', null, 'Hello, React!');
```

This React.createElement function call is what React uses under the hood to create a virtual DOM element.

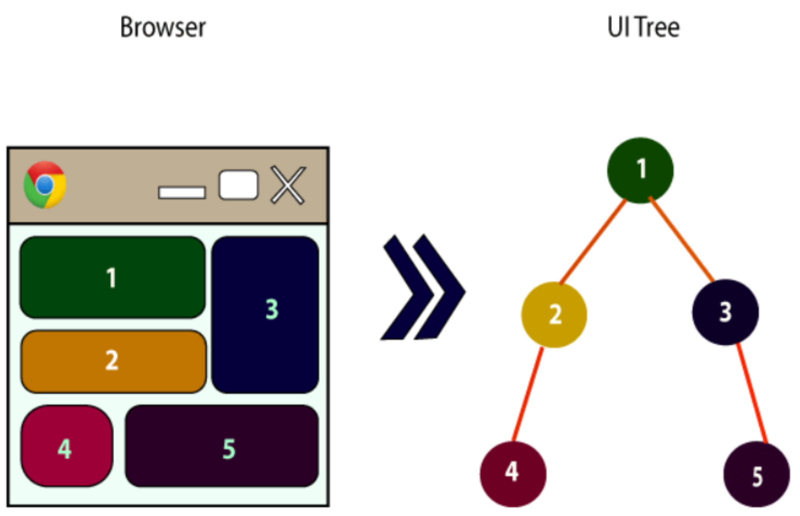
JSX makes the code more readable and closer to the structure of the actual UI, especially in the context of React components. It's important to note that JSX is not required to build React applications, but it is a popular choice among developers for its readability and conciseness.

## *React Components*

Earlier, the developers wrote 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 is found, it manually searches the entire application and updates 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 has their own structure,** methods as well as APIs. They can be reusable as per your need. For better understanding, consider the entire UI as a tree. Here, the root is the starting component, and each of the other pieces becomes branches, which are further divided into sub-branches.

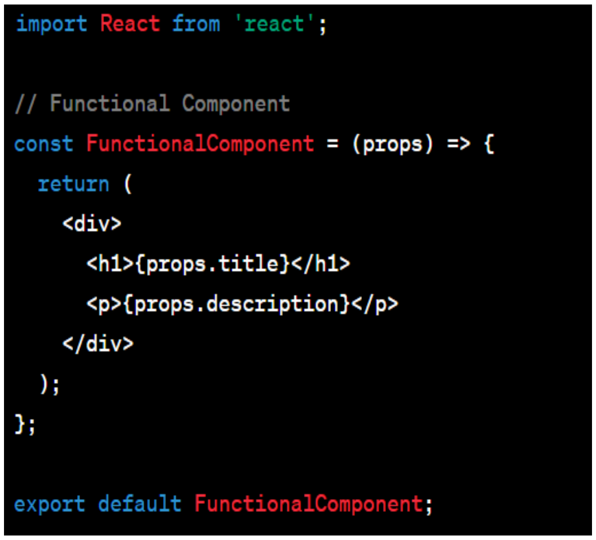


In ReactJS, we have mainly two types of components. They are

1. Functional Components
2. Class Components

## Functional Components

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

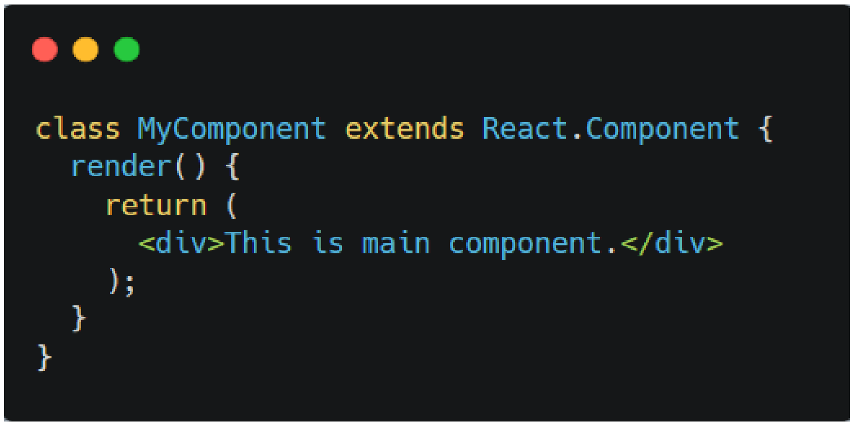


```
import React from 'react';

// Functional Component
const FunctionalComponent = (props) => {
  return (
    <div>
      <h1>{props.title}</h1>
      <p>{props.description}</p>
    </div>
  );
};

export default FunctionalComponent;
```

**NOTE:**The functional component is also known as a stateless component because they do not hold or manage state.

## 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>
    );
  }
}
```

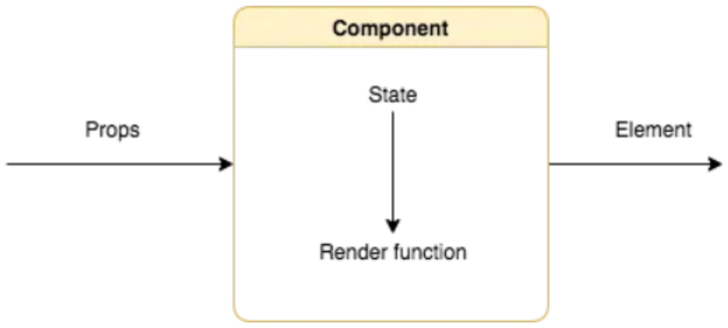
**NOTE:** The class component is also known as a stateful component because they can hold or manage local state.

## State and Props

In react components are responsible for generating html. To make html generate dynamically we need to pass data to our component so that our component can use variables and serve dynamic html.

There are 2 types of data in react:

- State (private data available in component only)
- Props (public data can be passed to component from outside)



## What are props in react?

Props are basically data that ***flows from one to another component*** as ***parameters***. Props can not be modified in a component.React Props are like function arguments in JavaScript *and* attributes in HTML. Components use this data to generate dynamic html elements.Props are passed to components via HTML attributes. Let's take a look at the following functional component example to understand.

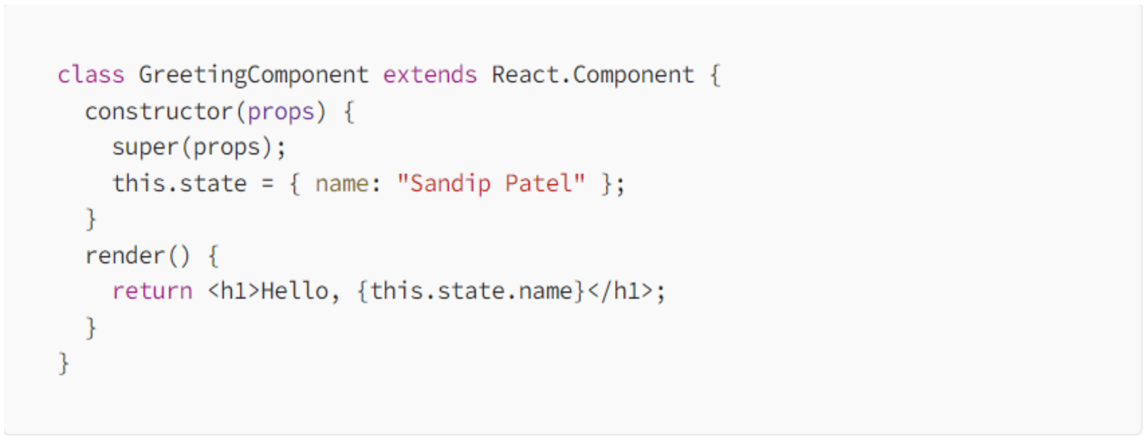
```
function GreetingComponent(props) {
  return <h1>Hello, {props.name}</h1>;
}


ReactDOM.render(
  <GreetingComponent name="Amisha" />,
  document.getElementById('root')
);
```

Using Class Components:

```
class GreetingComponent extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}

ReactDOM.render(
  <GreetingComponent name="Amisha" />,
  document.getElementById('root')
);
```

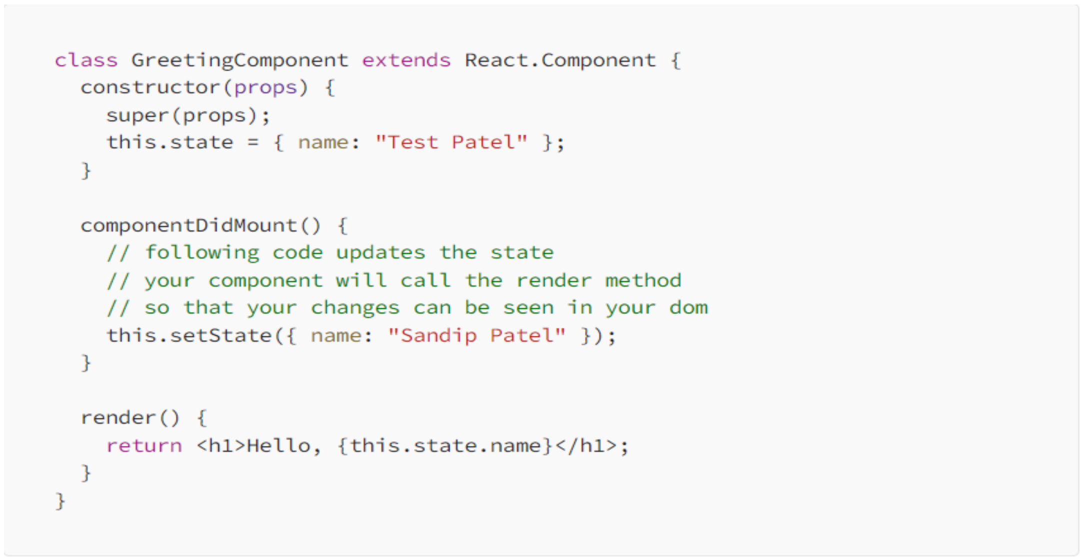
## What is state in react?

React components have a built-in state object which is private to a component. State can not be accessed from outside of the class. However it can be passed as an argument to another component.Whenever the state is changed, the component calls the render function to render html elements. Let's have a look at the following class component example:



```
class GreetingComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { name: "Sandip Patel" };
  }
  render() {
    return <h1>Hello, {this.state.name}</h1>;
  }
}
```

Updating a state variable:



```
class GreetingComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { name: "Test Patel" };
  }

  componentDidMount() {
    // following code updates the state
    // your component will call the render method
    // so that your changes can be seen in your dom
    this.setState({ name: "Sandip Patel" });
  }

  render() {
    return <h1>Hello, {this.state.name}</h1>;
  }
}
```

States in React are analogous to JavaScript objects, offering the flexibility to hold numerous properties. Here are examples illustrating how to modify the state:

Event Handling:

- Update the state in response to JavaScript events such as clicks or submits.

Component Mounting:

- Modify the state when a component mounts to the DOM by leveraging the componentDidMount method.

These approaches provide a structured means to manage and update the state in a React application.

## 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
-

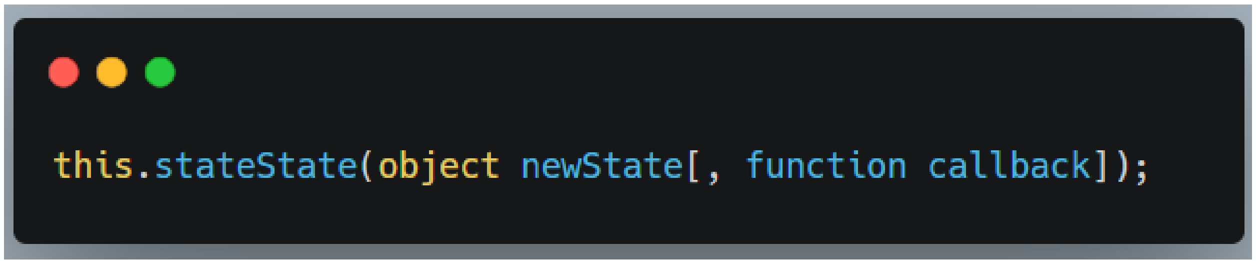
Here, we are going to explain the three most important methods available in the React component API.

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.

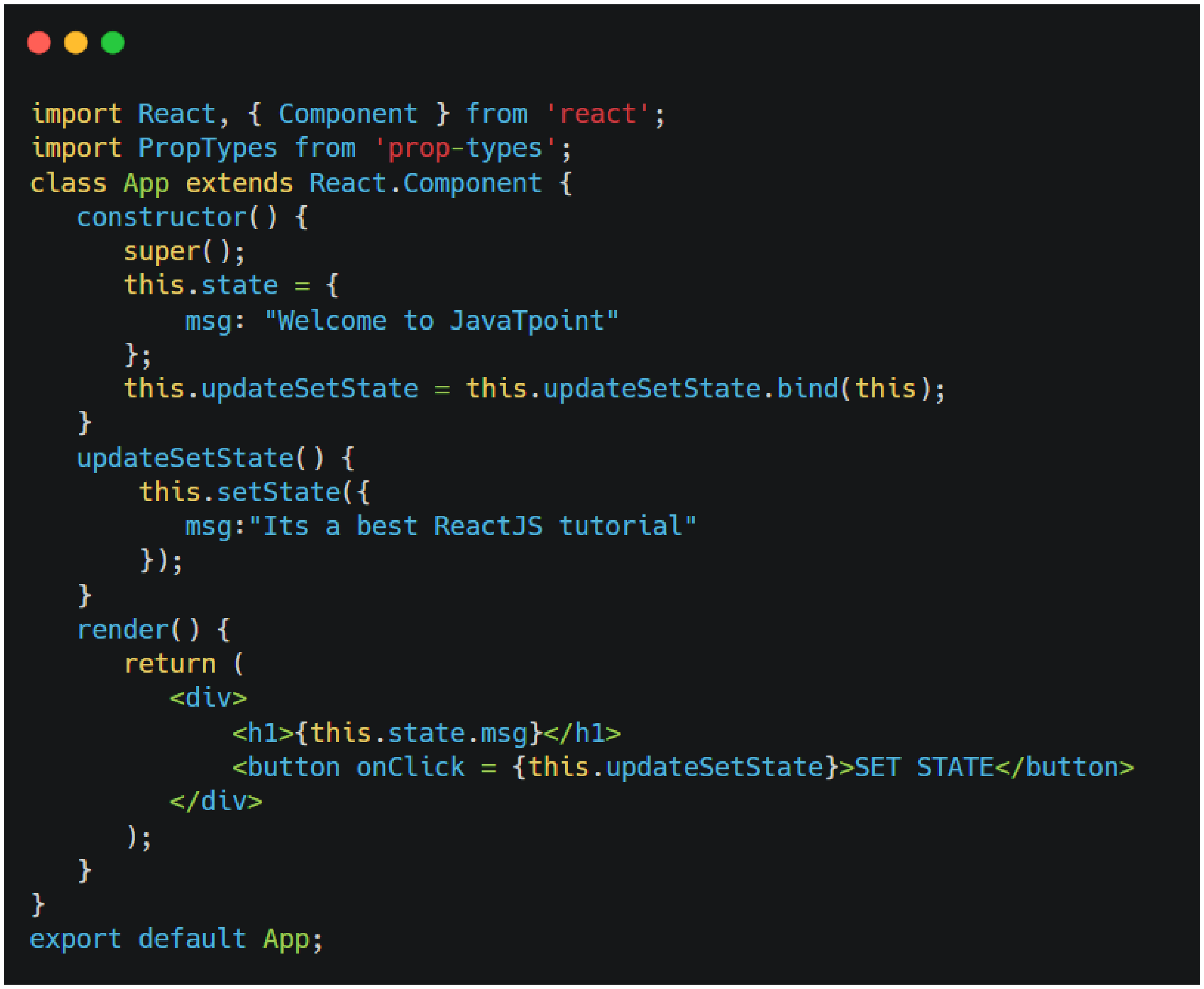
Syntax :



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

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

***Example***

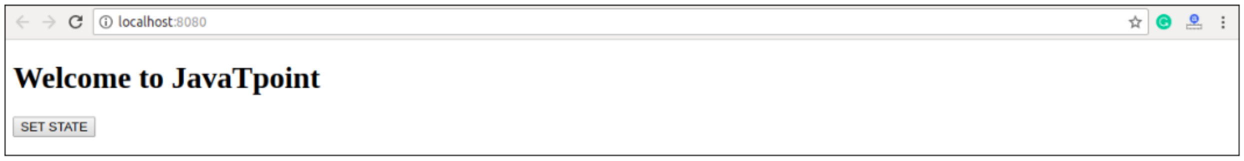


```
import React, { Component } from 'react';
import PropTypes from 'prop-types';
class App extends React.Component {
   constructor() {
      super();
      this.state = {
          msg: "Welcome to JavaTpoint"
      };
      this.updateSetState = this.updateSetState.bind(this);
   }
   updateSetState() {
       this.setState({
          msg:"Its a best ReactJS tutorial"
       });
   }
   render() {
      return (
         <div>
             <h1>{this.state.msg}</h1>
             <button onClick = {this.updateSetState}>SET STATE</button>
         </div>
      );
   }
}
export default App;
```

**Main.js**

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.js';

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

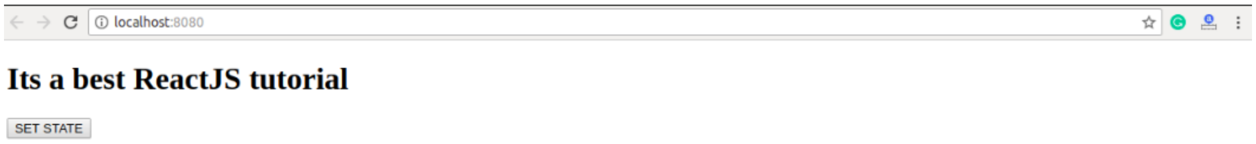
**Output:**


localhost:8080

**Welcome to JavaTpoint**

SET STATE

When you click on the **SET STATE** button, you will see the following screen with the updated message.


localhost:8080

**Its a best ReactJS tutorial**

SET STATE

## forceUpdate()

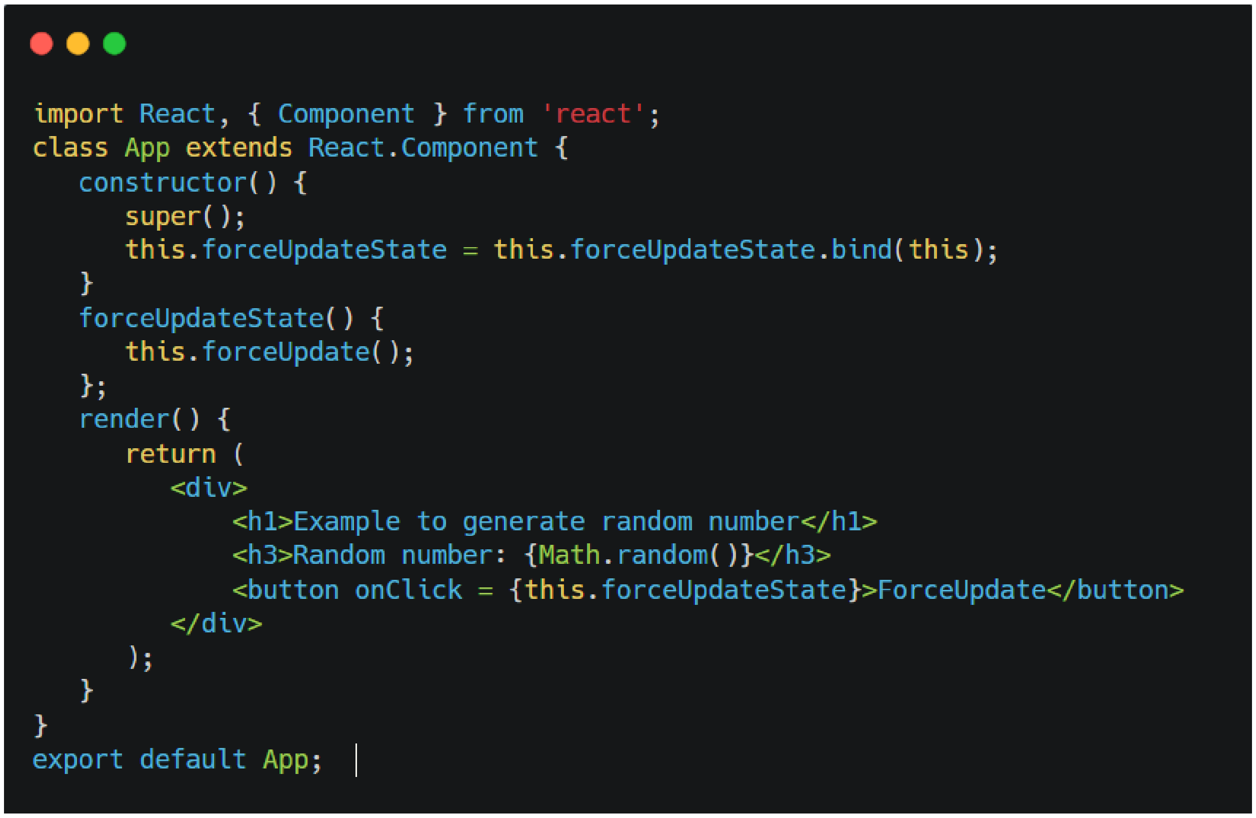
This method allows us to update the component manually.

### Syntax

```
Component.forceUpdate(callback);
```

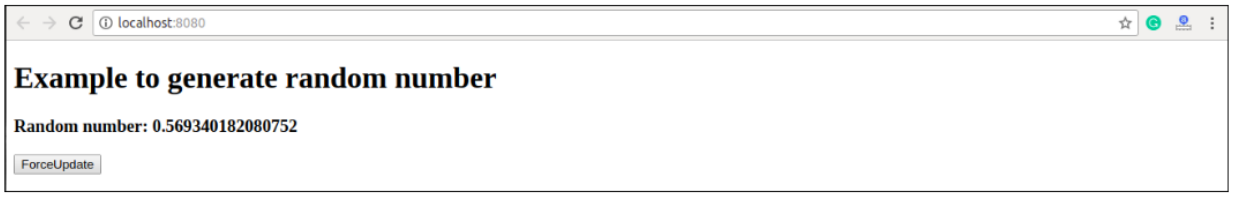
### Example

**App.js**



```
import React, { Component } from 'react';
class App extends React.Component {
   constructor() {
      super();
      this.forceUpdateState = this.forceUpdateState.bind(this);
   }
   forceUpdateState() {
      this.forceUpdate();
   };
   render() {
      return (
         <div>
             <h1>Example to generate random number</h1>
             <h3>Random number: {Math.random()}</h3>
             <button onClick = {this.forceUpdateState}>ForceUpdate</button>
         </div>
      );
   }
}
export default App;
```

**Output:**

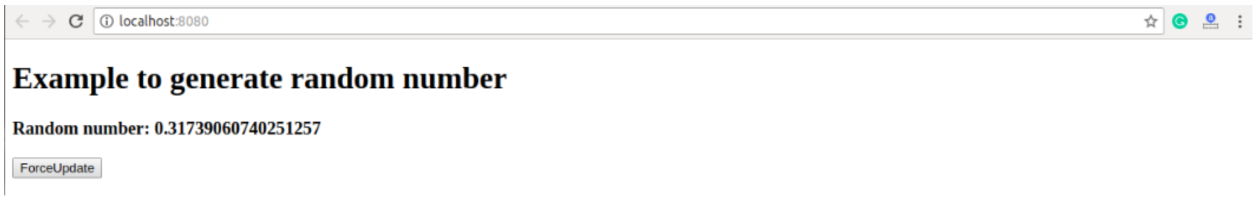


localhost:8080

**Example to generate random number**

**Random number: 0.569340182080752**

Each time when you click on ForceUpdate button, it will generate the random number. It can be shown in the below image.



localhost:8080

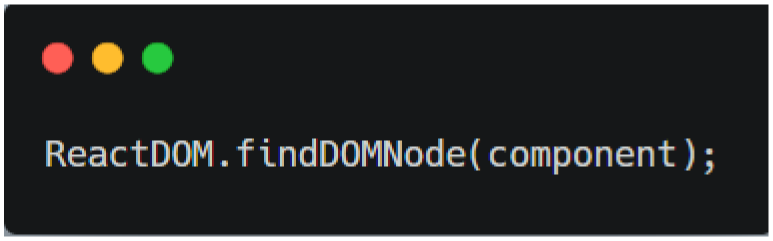
**Example to generate random number**

**Random number: 0.31739060740251257**

## findDOMNode()

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

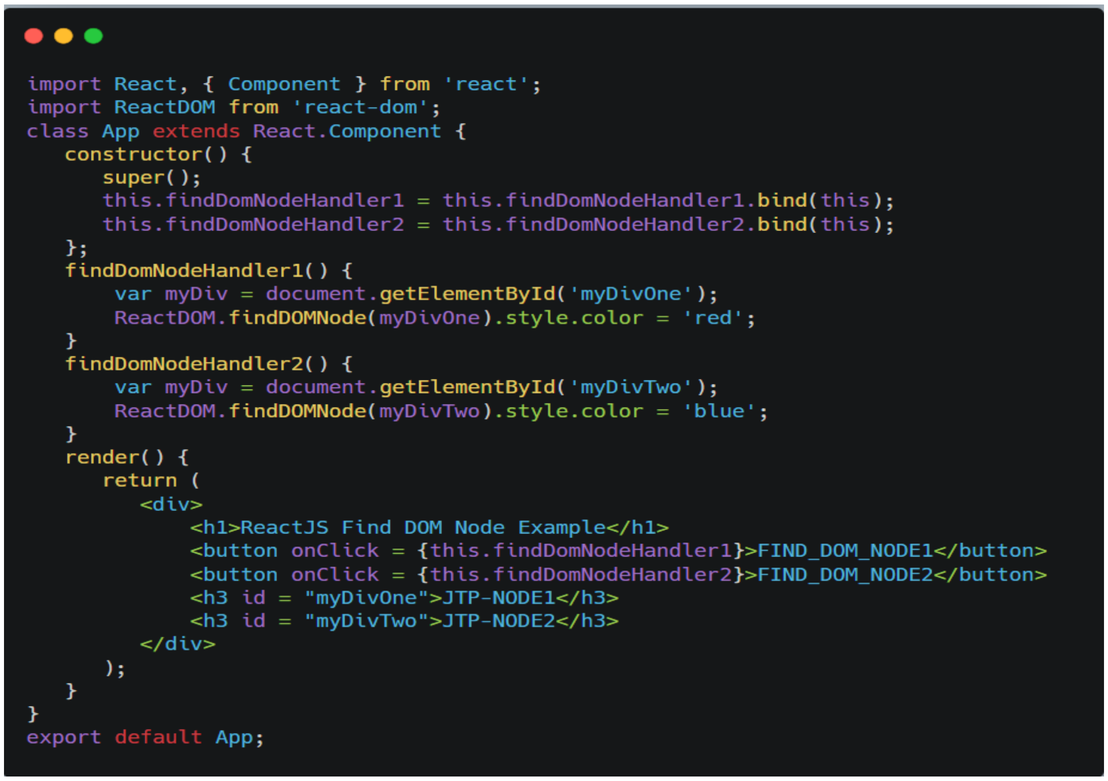
### Syntax



```
ReactDOM.findDOMNode(component);
```

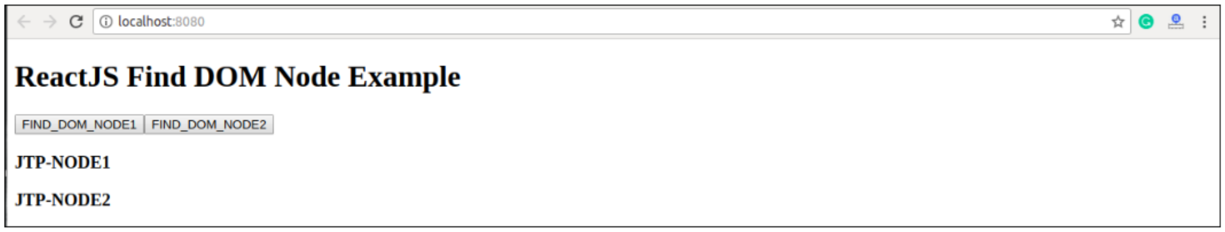
### Example

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>
      );
   }
}
export default App;
```

**Output:**



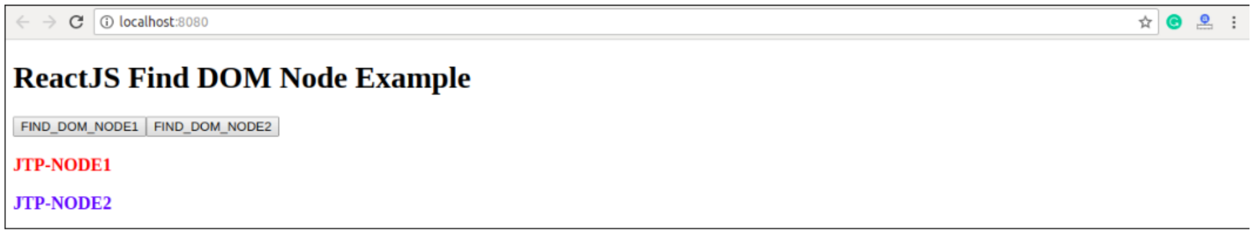
localhost:8080

ReactJS Find DOM Node Example

JTP-NODE1

JTP-NODE2

Once you click on the **button**, the color of the node gets changed. It can be shown in the below screen.



localhost:8080

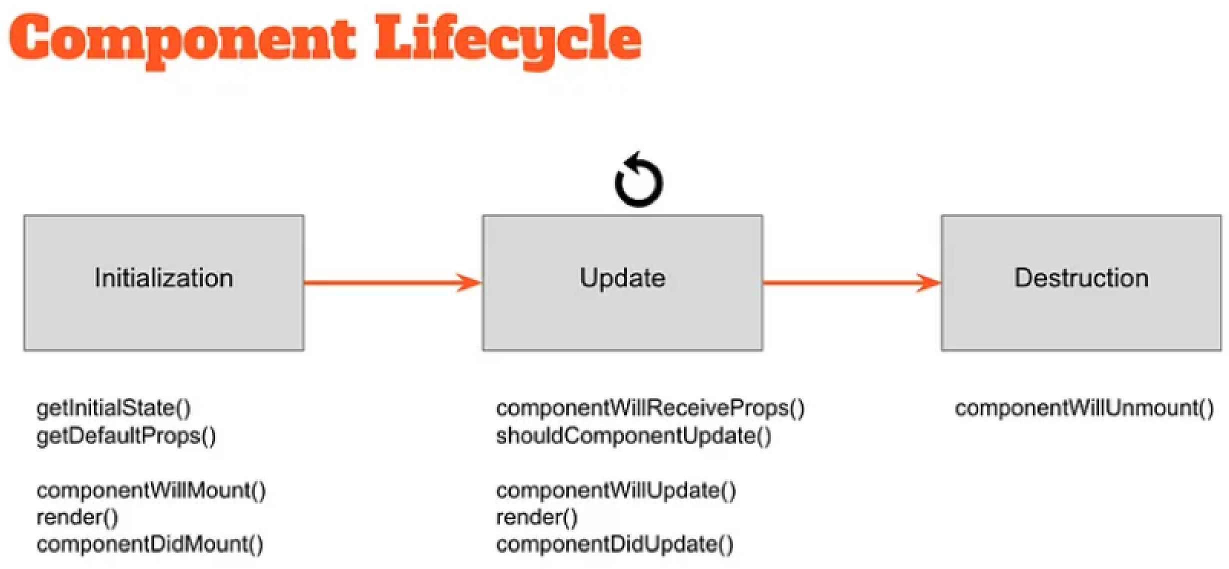
ReactJS Find DOM Node Example

JTP-NODE1

JTP-NODE2

## *React Component LifeCycle*

There are three stages of the component life cycle. Initialization/Mounting, State/Property Updates, and Destruction/Unmounting. There are React methods associated with each of these events and some of these events happen more than once. Understanding these three events can help you decide what logic to use in a given situation. For instance, we may want to add something to the DOM after a component has been rendered and then remove it after it right before it is destroyed.



All methods that have a `will` prefix are called before an action and all methods that have a `did` prefix are called right after an action

.

**`render()`**

This is arguably the most important method. This method is required in every component that is created. The `render` method should take into consideration `this.props` and `this.state` and return a react element (i.e. a DOM component as JSX or a user-created component in the form of `<MyComponent />` ), a string or numbers (which are created as text nodes in the DOM), `null` , or boolean values. `render()` should not modify the component's state and it should render predictably. `render()` does not get invoked if `shouldComponentUpdate()` returns `false`.

**Initialization/Mounting**

The first stage of the lifecycle is initialization/mounting. This is where the component is first being created and inserted into the DOM. The methods associated with this stage are shown above. I'm going to discuss the two below.

`componentWillMount()`

This method is invoked right before the component mounting occurs therefore before the render() method. If you are initially setting state it is recommended that you set that in the constructor() method instead. If you need to call setState() in this method it will not trigger any extra rendering since the component has not yet been rendered.

`componentDidMount()`

This method invokes as soon as component mounting occurs. Any thing that requires the presence of specific nodes on the DOM should go here. If there is data that needs to be requested from an API, this is a good place to put that request. If `setState()` is called here it will trigger a rerender since the initial render has already happened.

## State/Property Updating

Any time a change is made to the state or props a component is rerendered by default and there are other methods called on this rerender, some of which are explored below.

`shouldComponentUpdate()`

This method is used when letting a component know whether or not it should be updated based on changes in state or props. This is the first method invoked in the chain of methods that are associated with state/property changes.

`componentWillUpdate()`

This method is invoked before rendering occurs when there are changes to the state or props. This is where preparations can be done before an update. You should not make any changes to state or props in this method as it is assumed that changes to the state or props occurred already.

`componentDidUpdate()`

This method is invoked as soon as an update happens and is not called for the first render. This is a good place to make requests and check them against your state and props.

### Unmounting/Destroying

There is only one method in this category and it is called when a component is being removed from the DOM.

`componentWillUnmount()`

This method is called right before unmounting. This is where any cleanup occurs such as removing any timers, event listeners and stopping and network requests
.

# *Forms And Events*

## Forms

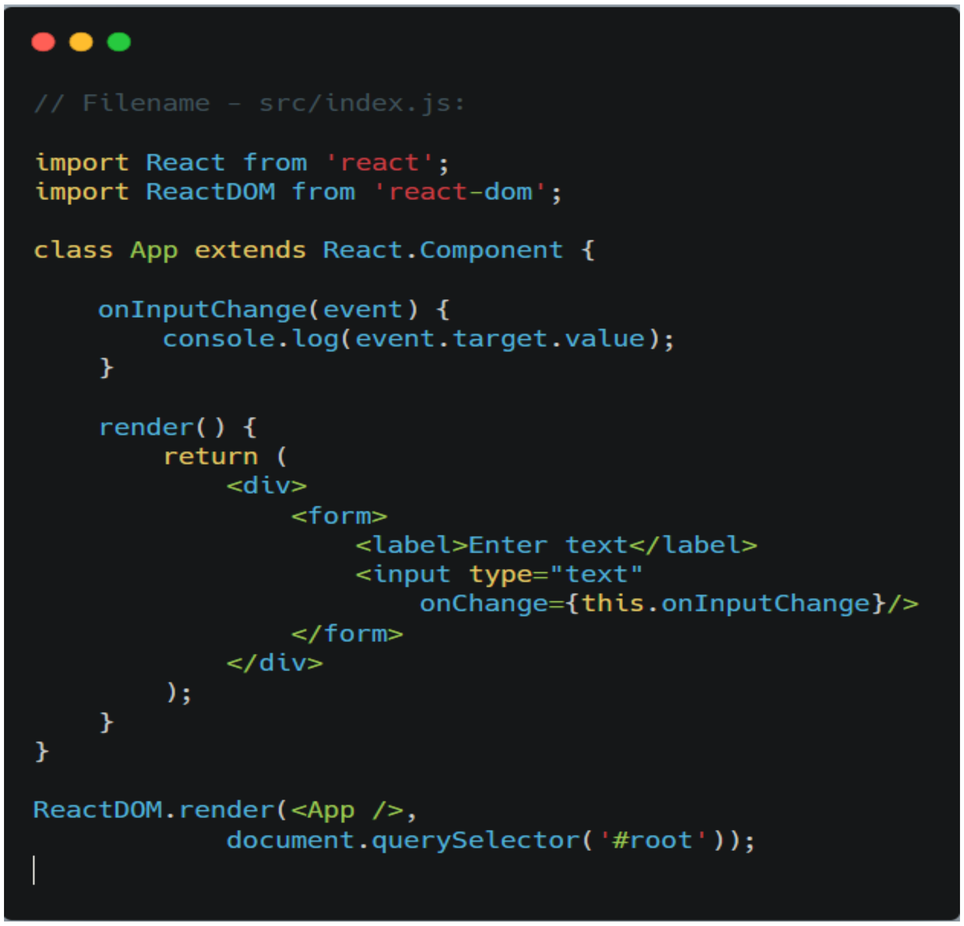
Forms are really important in any website for login, signup, or whatever. It is easy to make a form in HTML but forms in React work a little differently. In HTML the form data is usually handled by the DOM itself but in the case of react the form data is handled by the react components.

In React Forms, All the form data is stored in the React's component state, so it can handle the form submission and retrieve data that the user entered. To do this we use controlled components.
React uses the form to interact with the user and provides additional functionality such as preventing the default behavior of the form which refreshes the browser after the form is submitted.

### Controlled Components

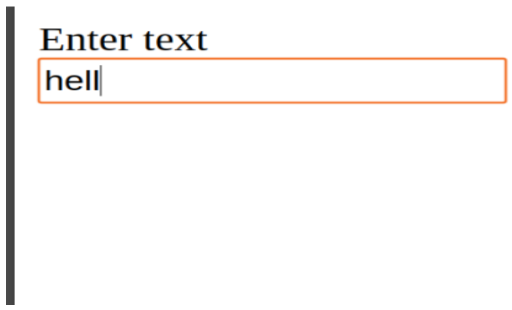
In simple HTML elements like input tags, the value of the input field is changed whenever the user type. But, In React, whatever the value the user types we save it in state and pass the same value to the input tag as its value, so here DOM does not change its value, it is controlled by react state. If you want to learn more about controlled components you can check the article Controlled Components in ReactJS. This may sound complicated But let's understand with an example.

**Example 1:** In this example code, we are going to console log the value of the input field with the help of the onInputChange function. and we'll see that every time the user input we get the output on the console of the browser.



```
// Filename - src/index.js:

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

class App extends React.Component {

    onInputChange(event) {
        console.log(event.target.value);
    }

    render() {
        return (
            <div>
                <form>
                    <label>Enter text</label>
                    <input type="text"
                        onChange={this.onInputChange}/>
                </form>
            </div>
        );
    }
}

ReactDOM.render(<App />,
            document.querySelector('#root'));
```

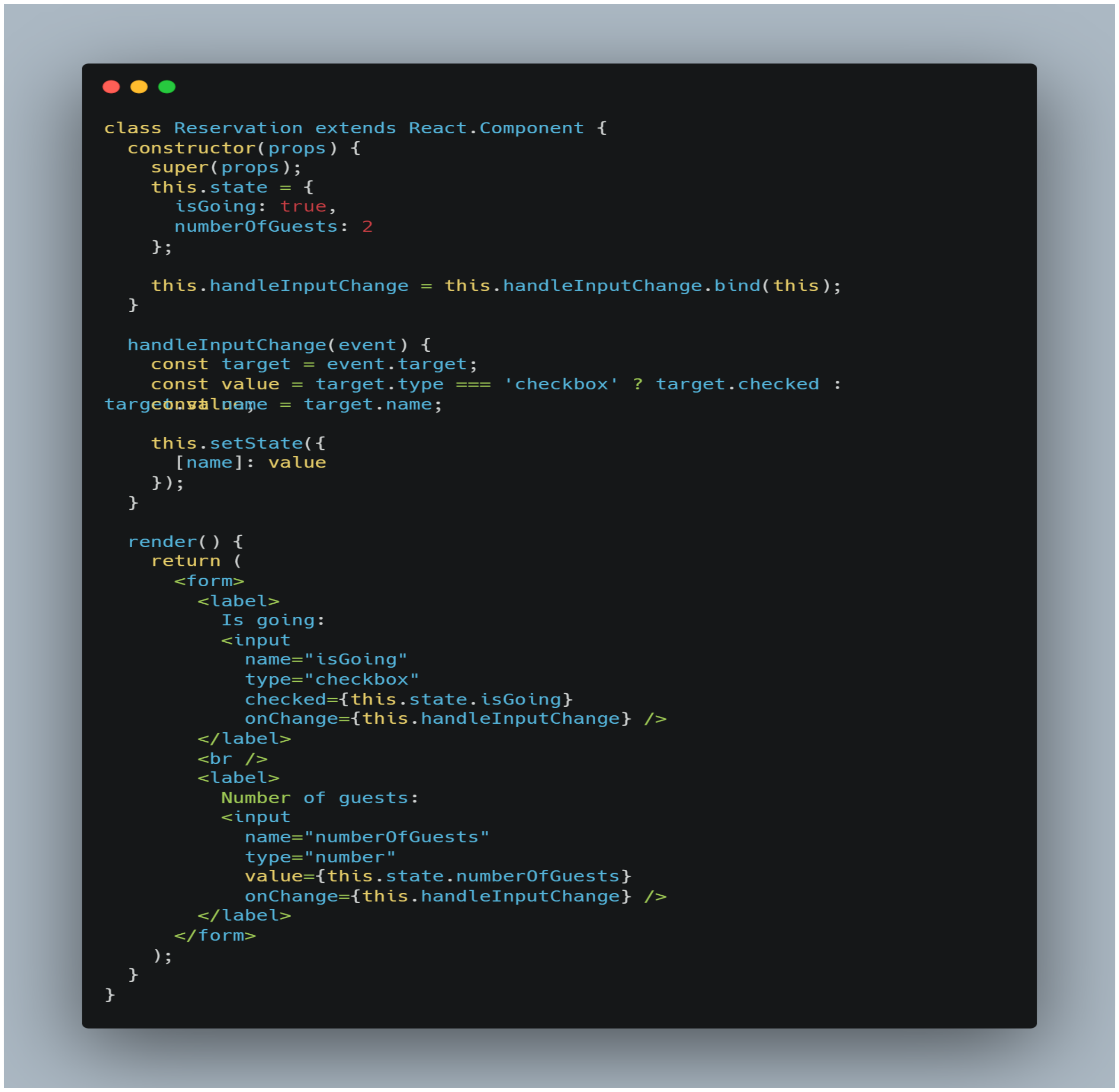
**Output:**



## Handling Multiple Inputs

When you need to handle multiple controlled `input` elements, you can add a `name` attribute to each element and let the handler function choose what to do based on the value of `event.target.name`.

For example:



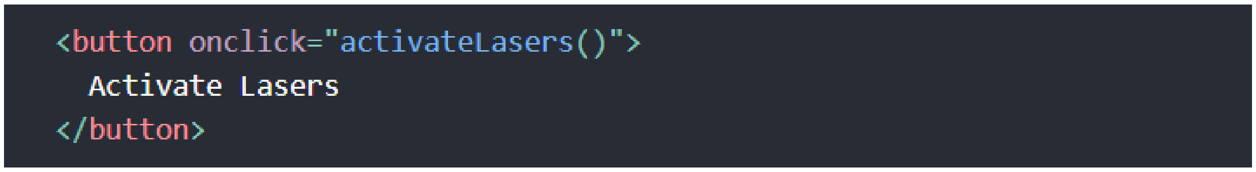
```
class Reservation extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isGoing: true,
      numberOfGuests: 2
    };

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

  handleInputChange(event) {
    const target = event.target;
    const value = target.type === 'checkbox' ? target.checked :
target.value;
    const name = target.name;

    this.setState({
      [name]: value
    });
  }

  render() {
    return (
      <form>
        <label>
          Is going:
          <input
            name="isGoing"
            type="checkbox"
            checked={this.state.isGoing}
            onChange={this.handleInputChange} />
        </label>
        <br />
        <label>
          Number of guests:
          <input
            name="numberOfGuests"
            type="number"
            value={this.state.numberOfGuests}
            onChange={this.handleInputChange} />
        </label>
      </form>
    );
  }
}
```

# Handling Events

Handling events with React elements is very similar to handling events on DOM elements. There are some syntax differences:

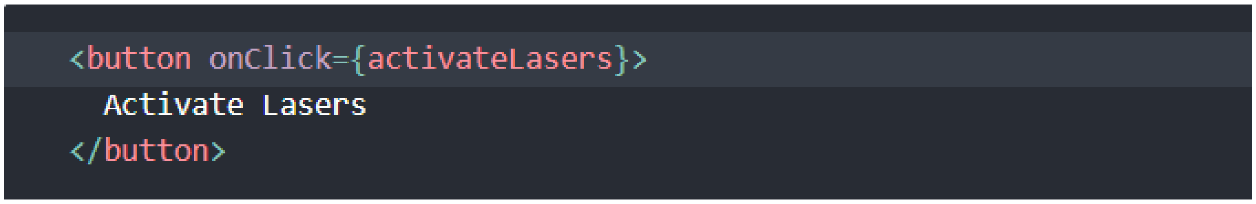
- React events are named using camelCase, rather than lowercase.
- With JSX you pass a function as the event handler, rather than a string.
-

For example, the HTML:



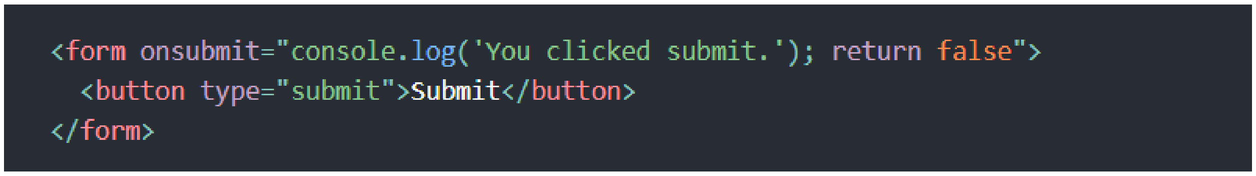
```html
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

is slightly different in React:



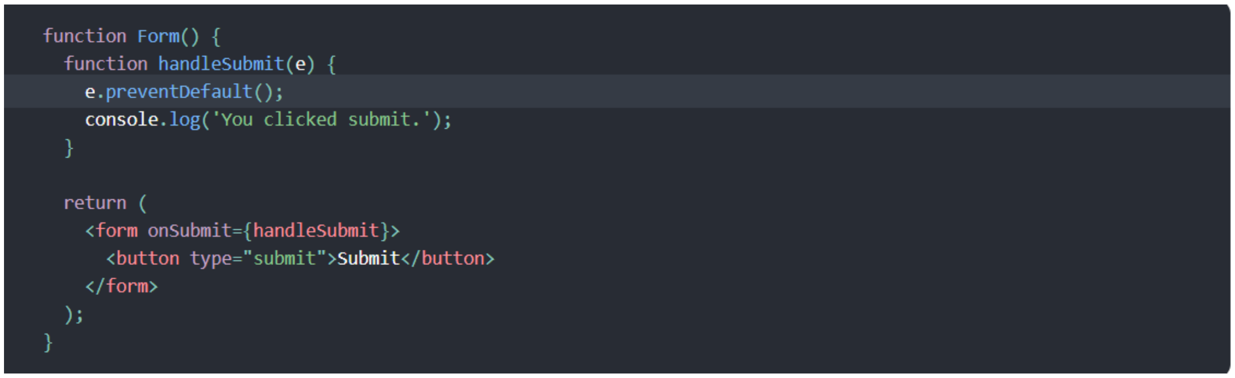
```jsx
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

Another difference is that you cannot return false to prevent default behavior in React. You must call preventDefault explicitly. For example, with plain HTML, to prevent the default form behavior of submitting, you can write:



```html
<form onsubmit="console.log('You clicked submit.'); return false">
  <button type="submit">Submit</button>
</form>
```

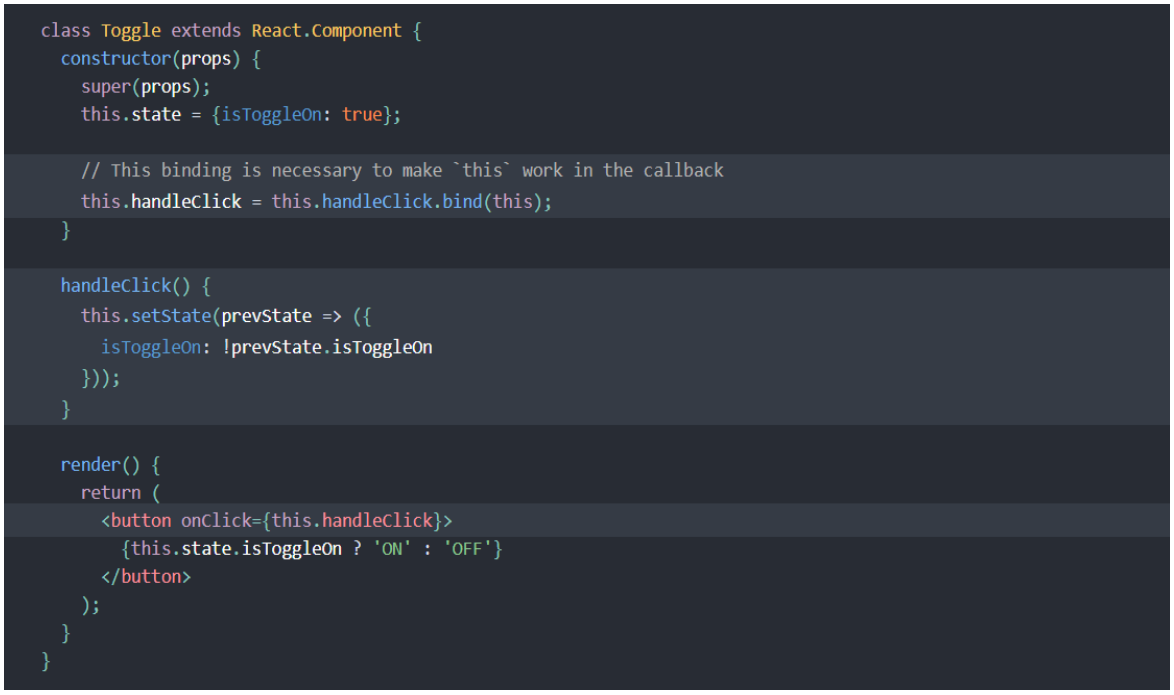
In React, this could instead be:



```
function Form() {
  function handleSubmit(e) {
    e.preventDefault();
    console.log('You clicked submit.');
  }

  return (
    <form onSubmit={handleSubmit}>
      <button type="submit">Submit</button>
    </form>
  );
}
```

Here, `e` is a synthetic event. React defines these synthetic events according to the W3C spec, so you don’t need to worry about cross-browser compatibility. React events do not work exactly the same as native events. When using React, you generally don’t need to call `addEventListener` to add listeners to a DOM element after it is created. Instead, just provide a listener when the element is initially rendered.

When you define a component using an ES6 class,a common pattern is for an event handler to be a method on the class. For example, this `Toggle` component renders a button that lets the user toggle between “ON” and “OFF” states:

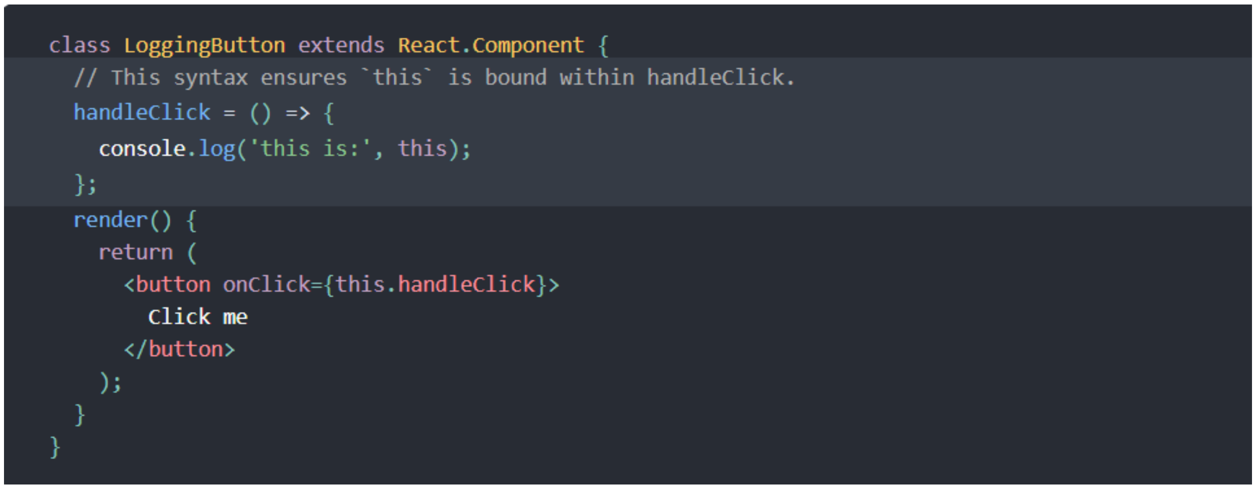


```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // This binding is necessary to make `this` work in the callback
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(prevState => ({
      isToggleOn: !prevState.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}
```

You have to be careful about the meaning of `this` in JSX callbacks. In JavaScript, class methods are not bound by default. If you forget to bind `this.handleClick` and pass it to `onClick`, `this` will be `undefined` when the function is actually called.

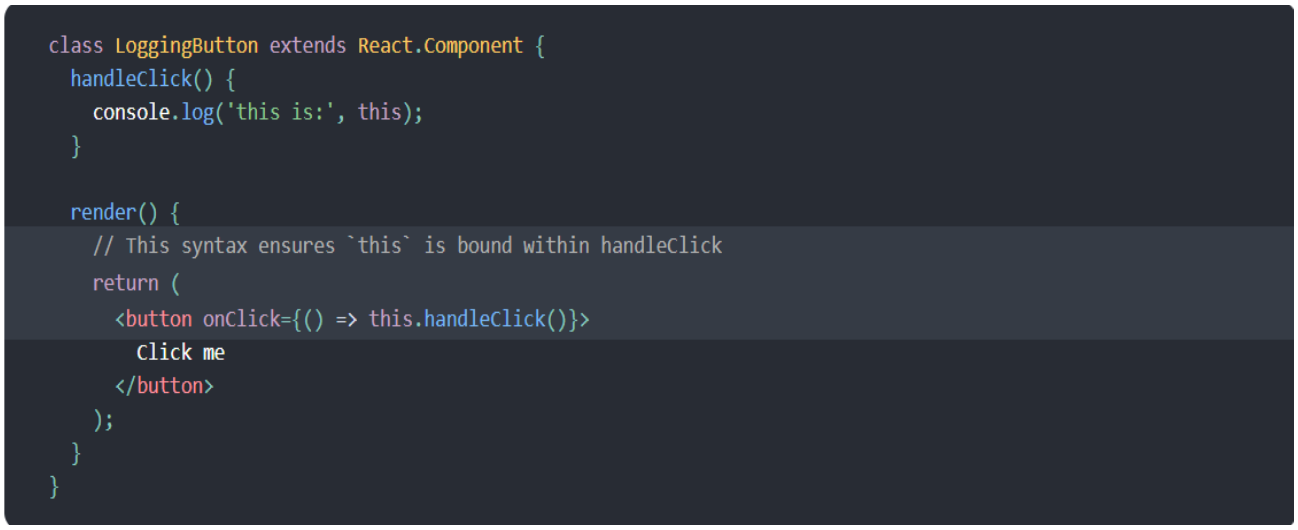
This is not React-specific behavior; it is a part of how functions work in JavaScript. Generally, if you refer to a method without () after it, such as `onClick={this.handleClick}`, you should bind that method.

If calling `bind` annoys you, there are two ways you can get around this. You can use public class fields syntax to correctly bind callbacks:



```
class LoggingButton extends React.Component {
  // This syntax ensures `this` is bound within handleClick.
  handleClick = () => {
    console.log('this is:', this);
  };
  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
```

This syntax is enabled by default in create React App.
If you aren’t using class fields syntax, you can use an arrow function in the callback:

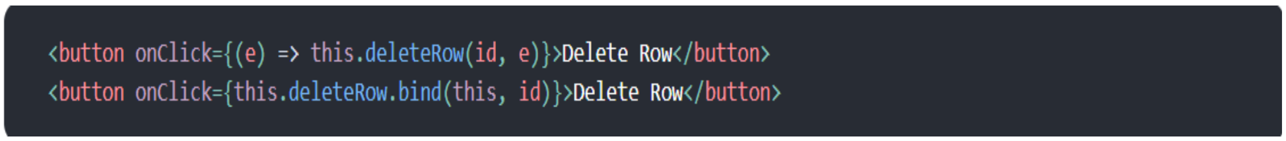


```
class LoggingButton extends React.Component {
  handleClick() {
    console.log('this is:', this);
  }

  render() {
    // This syntax ensures `this` is bound within handleClick
    return (
      <button onClick={() => this.handleClick()}>
        Click me
      </button>
    );
  }
}
```

The problem with this syntax is that a different callback is created each time the `LoggingButton` renders. In most cases, this is fine. However, if this callback is passed as a prop to lower components, those components might do an extra re-rendering. We generally recommend binding in the constructor or using the class fields syntax, to avoid this sort of performance problem.

## Passing Arguments to Event Handlers

Inside a loop, it is common to want to pass an extra parameter to an event handler. For example, if `id` is the row ID, either of the following would work:



```
<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>
<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>
```

The above two lines are equivalent, and use arrow functions and `Function.prototype.bind` respectively.

In both cases, the e argument representing the React event will be passed as a second argument after the ID. With an arrow function, we have to pass it explicitly, but with `bind` any further arguments are automatically forwarded.