Lifecycle of Components

Each component in React has a lifecycle which you can monitor and manipulate during its three main phases.

The three phases are: **Mounting**, **Updating**, and **Unmounting**.

Mounting

Mounting means putting elements into the DOM.

React has four built-in methods that gets called, in this order, when mounting a component:

1. `constructor()`
2. `getDerivedStateFromProps()`
3. `render()`
4. `componentDidMount()`

The `render()` method is required and will always be called, the others are optional and will be called if you define them.

constructor

The `constructor()` method is called before anything else, when the component is initiated, and it is the natural place to set up the initial `state` and other initial values.

The `constructor()` method is called with the `props`, as arguments, and you should always start by calling the `super(props)` before anything else, this will initiate the parent's constructor method and allows the component to inherit methods from its parent (`React.Component`).

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

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}
```

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

getDerivedStateFromProps

The `getDerivedStateFromProps()` method is called right before rendering the element(s) in the DOM.

This is the natural place to set the `state` object based on the initial `props`.

It takes `state` as an argument, and returns an object with changes to the `state`.

The example below starts with the favorite color being "red", but the `getDerivedStateFromProps()` method updates the favorite color based on the `favcol` attribute:

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

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  static getDerivedStateFromProps(props, state) {
    return {favoritecolor: props.favcol };
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header favcol="yellow"/>);
```

render

The `render()` method is required, and is the method that actually outputs the HTML to the DOM.

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

class Header extends React.Component {
  render() {
    return (
      <h1>This is the content of the Header component</h1>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

componentDidMount

The `componentDidMount()` method is called after the component is rendered.

This is where you run statements that requires that the component is already placed in the DOM.

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

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoritecolor: "yellow"})
    }, 1000)
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

Updating

The next phase in the lifecycle is when a component is *updated*.

A component is updated whenever there is a change in the component's `state` or `props`.

React has five built-in methods that gets called, in this order, when a component is updated:

1. `getDerivedStateFromProps()`
2. `shouldComponentUpdate()`
3. `render()`
4. `getSnapshotBeforeUpdate()`
5. `componentDidUpdate()`

The `render()` method is required and will always be called, the others are optional and will be called if you define them.

getDerivedStateFromProps

Also at *updates* the `getDerivedStateFromProps` method is called. This is the first method that is called when a component gets updated.

This is still the natural place to set the `state` object based on the initial props.

The example below has a button that changes the favorite color to blue, but since the `getDerivedStateFromProps()` method is called, which updates the state with the color from the favcol attribute, the favorite color is still rendered as yellow:

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

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  static getDerivedStateFromProps(props, state) {
    return {favoritecolor: props.favcol };
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
  render() {
    return (
      <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <button type="button" onClick={this.changeColor}>Change color</button>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header favcol="yellow" />);
```

shouldComponentUpdate

In the `shouldComponentUpdate()` method you can return a Boolean value that specifies whether React should continue with the rendering or not.

The default value is `true`.

The example below shows what happens when the `shouldComponentUpdate()` method returns `false`:

```
import React from 'react';

class ComponentUpdate extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  shouldComponentUpdate() {
    return false;
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
  render() {
    return (
      <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <button type="button" onClick={this.changeColor}>Change color</button>
      </div>
    );
  }
}

export default ComponentUpdate;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);

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

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  shouldComponentUpdate() {
    return true;
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
  render() {
    return (
      <div>render
```

The `render()` method is of course called when a component gets *updated*, it has to re-render the HTML to the DOM, with the new changes.

The example below has a button that changes the favorite color to blue:

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

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
  render() {
    return (
      <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <button type="button" onClick={this.changeColor}>Change color</button>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

getSnapshotBeforeUpdate

In the `getSnapshotBeforeUpdate()` method you have access to the `props` and `state` *before* the update, meaning that even after the update, you can check what the values were *before* the update.

If the `getSnapshotBeforeUpdate()` method is present, you should also include the `componentDidUpdate()` method, otherwise you will get an error.

The example below might seem complicated, but all it does is this:

When the component is *mounting* it is rendered with the favorite color "red".

When the component *has been mounted,* a timer changes the state, and after one second, the favorite color becomes "yellow".

This action triggers the *update* phase, and since this component has a `getSnapshotBeforeUpdate()` method, this method is executed, and writes a message to the empty DIV1 element.

Then the `componentDidUpdate()` method is executed and writes a message in the empty DIV2 element:

```
import React from 'react';

class SnapShot extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoritecolor: "yellow"})
    }, 5000)
  }
  getSnapshotBeforeUpdate(prevProps, prevState) {
    document.getElementById("div1").innerHTML =
    "Before the update, the favorite was " + prevState.favoritecolor;
  }
  componentDidUpdate() {
    document.getElementById("div2").innerHTML =
    "The updated favorite is " + this.state.favoritecolor;
  }
  render() {
    return (
      <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <div id="div1"></div>
      <div id="div2"></div>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

componentDidUpdate

The `componentDidUpdate` method is called after the component is updated in the DOM.

The example below might seem complicated, but all it does is this:

When the component is *mounting* it is rendered with the favorite color "red".

When the component *has been mounted,* a timer changes the state, and the color becomes "yellow".

This action triggers the *update* phase, and since this component has a `componentDidUpdate` method, this method is executed and writes a message in the empty DIV element:

```
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <button type="button" onClick={this.changeColor}>Change color</button>
```

```
    </div>
  );
 }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

componentDidUpdate

The `componentDidUpdate` method is called after the component is updated in the DOM.

The example below might seem complicated, but all it does is this:

When the component is *mounting* it is rendered with the favorite color "red".

When the component *has been mounted,* a timer changes the state, and the color becomes "yellow".

This action triggers the *update* phase, and since this component has a `componentDidUpdate` method, this method is executed and writes a message in the empty DIV element:

```
import React from 'react';

class ComponentUpdate extends React.Component {
 constructor(props) {
  super(props);
  this.state = {favoritecolor: "red"};
 }
 componentDidMount() {
  setTimeout(() => {
   this.setState({favoritecolor: "yellow"})
  }, 5000)
 }
 componentDidUpdate() {
  document.getElementById("mydiv").innerHTML =
  "The updated favorite is " + this.state.favoritecolor;
 }
 render() {
  return (
   <div>
   <h1>My Favorite Color is {this.state.favoritecolor}</h1>
```

```
      <div id="mydiv"></div>
      </div>
    );
  }
}

export default ComponentUpdate;


const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

Unmounting

The next phase in the lifecycle is when a component is removed from the DOM, or *unmounting* as React likes to call it.

React has only one built-in method that gets called when a component is unmounted:

- `componentWillUnmount()`

componentWillUnmount

The `componentWillUnmount` method is called when the component is about to be removed from the DOM.

```
import React from 'react';

class Container extends React.Component {
  constructor(props) {
    super(props);
    this.state = {show: true};
  }
  delHeader = () => {
    this.setState({show: false});
  }
  render() {
    let myheader;
    if (this.state.show) {
      myheader = <Child />;
    };
    return (
      <div>
      {myheader}
      <button type="button" onClick={this.delHeader}>Delete Header</button>
      </div>
    );
  }
}
class Child extends React.Component {
  componentWillUnmount() {
    alert("The component named Header is about to be unmounted.");
  }
  render() {
    return (
      <h1>Hello World!</h1>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Container />);
```

React Class Component Example 1

In this example, we define a class component called `Employee`. It extends the `Component` class from the react package. The component has an initial state object with properties firstName, lastName, and email:

```
import React from "react"

class Employee extends React.Component {
    constructor(props) {
        super(props)

        this.state = {
            firstName: "Ramesh",
            lastName: "Fadatare",
            email: "ramesh@gmail.com"
        }
    }

    render(){
        return (
            <div className="center">
                <h1> Employee Details</h1> <hr />
                <p>{this.state.firstName}</p>
                <p>{this.state.lastName}</p>
                <p>{this.state.email}</p>
            </div>
        )
    }
}

export default Employee
```

Next, use this `Employee` class component in the App component:

```
import Greeting from './components/Employee'

function App() {

  return (
    <div>
        <Employee />
    </div>
  )
}

export default App
```

React Class Component Example 2

In this example, we define a class component called `Counter`. It extends the `Component` class from the react package. The component has an initial state with a count property set to 0.

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

class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  handleClick = () => {
    this.setState({ count: this.state.count + 1 });
  };

  render() {
    return (
      <div>
```

```
        <h2>Count: {this.state.count}</h2>
        <button onClick={this.handleClick}>Increment</button>
      </div>
    );
  }
}

export default Counter;
```

Next, use this Counter class component in the `App` component:

```
import Greeting from './components/Counter'

function App() {

  return (
    <div>
        <Counter />
    </div>
  )
}

export default App
```

The `handleClick` method is defined as an arrow function within the class component. When the button is clicked, it calls this method to update the count state by incrementing it by 1 using setState().

The `render()` method is required in a class component and returns the JSX that defines the component's UI. It displays the current count value and a button to increment the count.

Basic Functional Component

The functional component is basically a JavaScript/ES6 arrow function that returns a React element (JSX).

In this example, we define a basic functional component called `Greeting` that renders a simple greeting message.

```
import React from 'react';

function Greeting() {
  return <h1>Hello, World!</h1>;
};
```

```
export default Greeting;
```

Let's rewrite the above functional component using the ES6 arrow function:

```
import React from 'react';

const Greeting = () => {
  return <h1>Hello, World!</h1>;
};

export default Greeting;
```

Functional Component with Props

Here, we define a functional component called `Welcome` that accepts a prop called name and renders a personalized welcome message.

```
import React from 'react';

const Welcome = (props) => {
  return <h1>Welcome, {props.name}!</h1>;
};

export default Welcome;
```

Next, use the `Welcome` functional component in the `App` component and pass the `name` property:

```
import Welcome from './components/Welcome'

function App() {

  return (
    <div>
        <Welcome name = "Ramesh"/>
    </div>
  )
}

export default App
```

Functional Component with JSX and JavaScript Expressions

In this example, we define a functional component called `Counter` that calculates and displays the count and its doubled value using JavaScript expressions within JSX.

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

const ButtonCounter = () => {
  const [count, setCount] = useState(0);

  const handleClick = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={handleClick}>Increment</button>
    </div>
  );
};

export default ButtonCounter;
```

Functional Component with Event Handling

Here, we define a functional component called `ButtonCounter` that tracks a count state using the `useState` hook. It renders the count value and a button that increments the count when clicked, using an event handler function.

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

const ButtonCounter = () => {
  const [count, setCount] = useState(0);

  const handleClick = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={handleClick}>Increment</button>
    </div>
  );
};

export default ButtonCounter;
```

React Props Example 1

The below code example demonstrates the usage of `props` in React to pass data from a parent component to a child component.

```
export const Student = (props) => {

    console.log(props);
    return (
        <div className="center">
            <p>First name: {props.firstName}</p>
            <p>Last name: {props.lastName}</p>
            <p>Email Address: {props.email}</p>
        </div>
    )
}
```

Inside the component's JSX, the `props` values are accessed and displayed. It shows the first name, last name, and email address of a student.

Next, import and use the `Student` component in an `App` component:

```
import { Student } from './components/PropsDemo'

function App() {
  return (
    <div>
       <Student
          firstName = "Ramesh"
          lastName="Fadatare"
          email="ramesh@gmail.com"
    />

    <Student
      firstName = "Umesh"
      lastName="Fadatare"
      email="umesh@gmail.com"
    />

   </div>
  )
}

export default App
```

When the `App` component is rendered, it includes two instances of the `Student` component. Each instance receives different values for the firstName, lastName, and email props. The props are then accessed and displayed in the `Student` component's JSX, showing the student's information on the rendered page.

React Props Example 2 - Pass an object as props to a React component

We may pass an object as `props` to a React component.

```
export const Student = (props) => {

   console.log(props);
   return (
     <div className="center">
       <p>First name: {props.student.firstName}</p>
       <p>Last name: {props.student.lastName}</p>
       <p>Email Address: {props.student.email}</p>
     </div>
   )
}
```

App component:

```
import { Student } from './components/PropsDemo'

function App() {

  const student = {
    firstName : "Ramesh",
    lastName : "Fadatare",
    email : "ramesh@gmail.com"
  }

  return (
    <div>
      <Student student={ student }/>
    </div>
  )
}

export default App
```

React Props Example 3 - Pass an Array as props to a React component

We may pass an array as props to a React component.

```
export const Student = (props) => {

    console.log(props);
    return (
        <div className="center">
            <p> Array data: {props.data} </p>
        </div>
    )
}
```

App component:

```
import { Student } from './components/PropsDemo'

function App() {


  const skills = ['HTML', 'CSS', 'JavaScript']

  return (
    <div>
      <Student data = { skills }/>
    </div>
  )
}

export default App
```

Destructuring props

Destructuring was introduced in ES6. It’s a JavaScript feature that allows us to extract multiple pieces of data from an array or object and assign them to their own variables.

In React, destructuring props improves code readability.

Two ways to destructure props in functional component

There are two ways to destructure props in a react component. The first way is destructuring it in the function parameter itself:

```
export const Student = ({firstName, lastName, email}) => {

    return (
        <div className="center">
            <p>First name: {firstName}</p>
            <p>Last name: {lastName}</p>
            <p>Email Address: {email}</p>
        </div>
    )
}
```

App Component:

```
import { useState } from 'react'
import reactLogo from './assets/react.svg'
import './App.css'
import { Student } from './components/PropsDemo'

function App() {

  return (
    <div>
      <Student
          firstName = "Ramesh"
          lastName="Fadatare"
          email="ramesh@gmail.com"
      />

      <Student
          firstName = "Umesh"
          lastName="Fadatare"
          email="umesh@gmail.com"
      />
    </div>
  )
}

export default App
```

The second way is destructuring props in the function body:

```
export const Student = (props) => {
    const {firstName, lastName, email} = props

    return (
        <div className="center">
            <p>First name: {firstName}</p>
            <p>Last name: {lastName}</p>
            <p>Email Address: {email}</p>
        </div>
    )
}
```

A state is a built-in object in React class components. In the state object, we store property values that belong to the component. When the state object changes, the component re-renders. We use the `setState()` method to change the state object in a class component.

Example 1: State and setState in Class Components

Create State Object in Employee Component

Let's create an `Employee` class component and create and initialize the state object in the constructor:

```
import React from "react"

class Employee extends React.Component {
    constructor(props) {
        super(props)

        this.state = {
            firstName: "Ramesh",
            lastName: "Fadatare",
            email: "ramesh@gmail.com"
        }
    }

    render(){
        return (
            <div className="center">
                <h1> Employee Details</h1> <hr />
                <p>{this.state.firstName}</p>
                <p>{this.state.lastName}</p>
                <p>{this.state.email}</p>
            </div>
        )
    }
}

export default Employee
```

Let's import the above component into the `App` component and see the result in the browser:

```
import './App.css'
import Employee from './components/Employee'

function App() {
  return (
    <div>
      <Employee />
    </div>
  )
}

export default App
```

Using setState() Method to Update the State Object

To change a value in the state object, use `this.setState()` method. When a value in the state object changes, the component will re-render, meaning that the output will change according to the new value(s).

For example, let's add a button with an onClick event that will change the lastName and email properties of the employee:

```
import React from "react"

class Employee extends React.Component {
    constructor(props) {
        super(props)

        this.state = {
            firstName: "Ramesh",
            lastName: "Fadatare",
            email: "ramesh@gmail.com"
        }
    }

    updateEmployee(){
        this.setState({
            lastName: "jadhav",
            email: "ram@gmail.com"
        })
    }

    render(){
        return (
            <div className="center">
                <h1> Employee Details</h1> <hr />
                <p>{this.state.firstName}</p>
                <p>{this.state.lastName}</p>
                <p>{this.state.email}</p>

                <button onClick={() => this.updateEmployee()}>Update Employee</
button>
            </div>
        )
    }
}

export default Employee
```

Example 2: state and setState()

Creating the state Object

Let's create a `StudentComponent` and create and initialize the `state` object in the constructor:

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

class StudentComponent extends Component {
    constructor(props) {
        super(props)

        this.state = {
                firstName: "Ramesh"
        }
    }

    render() {
        return (
            <div>
                <h1> Hello Student</h1>
            </div>
        )
    }
}

export default StudentComponent
```

The `state` object can contain as many properties. For example, let's specify all the properties your component need:

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

class StudentComponent extends Component {
    constructor(props) {
        super(props)

        this.state = {
                firstName: "Ramesh",
                lastName:"Fadatare",
                rollNo: 123,
                age: 20,
                books: ["C programming", "C++ programming",
"Data Structure and Algorithms"]
        }
    }

    render() {
        return (
            <div>
                <h1> Hello Student</h1> <hr/>
            </div>
        )
    }
}

export default StudentComponent
```

Now, we have seen how to create `state` and initiate the `state` object in a component.

Let's see how to use `state` object in the component.

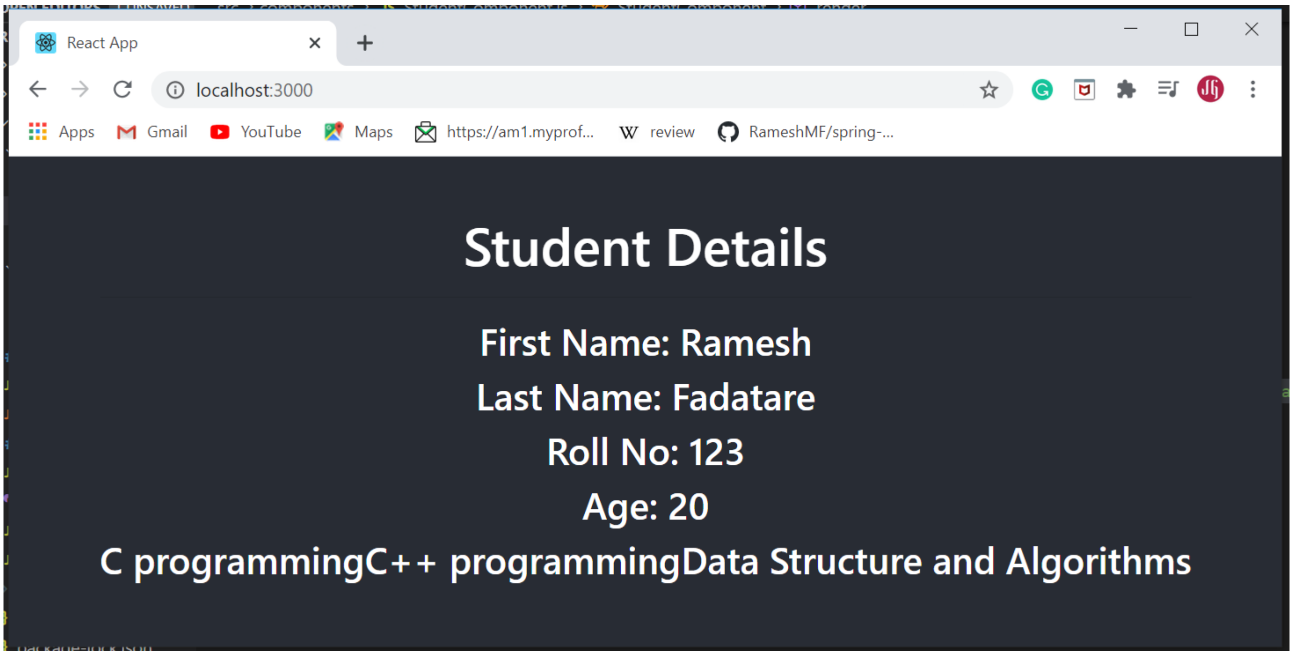
Using the state Object

Refer to the `state` object anywhere in the component by using the following syntax:

```
this.state.propertyname
```

Example: Refer to the `state` object in the `render()` method:

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

class StudentComponent extends Component {
    constructor(props) {
        super(props)

        this.state = {
                firstName: "Ramesh",
                lastName:"Fadatare",
                rollNo: 123,
                age: 20,
                books: ["C programming", "C++ programming",
"Data Structure and Algorithms"]
        }
    }

    render() {
        return (
            <div>
                <h1> Student Details</h1> <hr/>
                <h3> First Name: {this.state.firstName }</h3>
                <h3> Last Name: {this.state.lastName }</h3>
                <h3> Roll No: {this.state.rollNo } </h3>
                <h3> Age: {this.state.age }</h3>
                <h3> {this.state.books} </h3>
            </div>
        )
    }
}

export default StudentComponent
```
Let's import above component into the `App` component and see the result in the browser:
```
import React from 'react';
import logo from './logo.svg';
import './App.css';
import StudentComponent from './components/StudentComponent';

function App() {
  return (
    <div className="App">
         <header className="App-header">
            <StudentComponent />
        </header>
    </div>
  );
}

export default App;
```
Hit URL http://localhost:3000/ in Browser



Student Details

First Name: Ramesh

Last Name: Fadatare

Roll No: 123

Age: 20

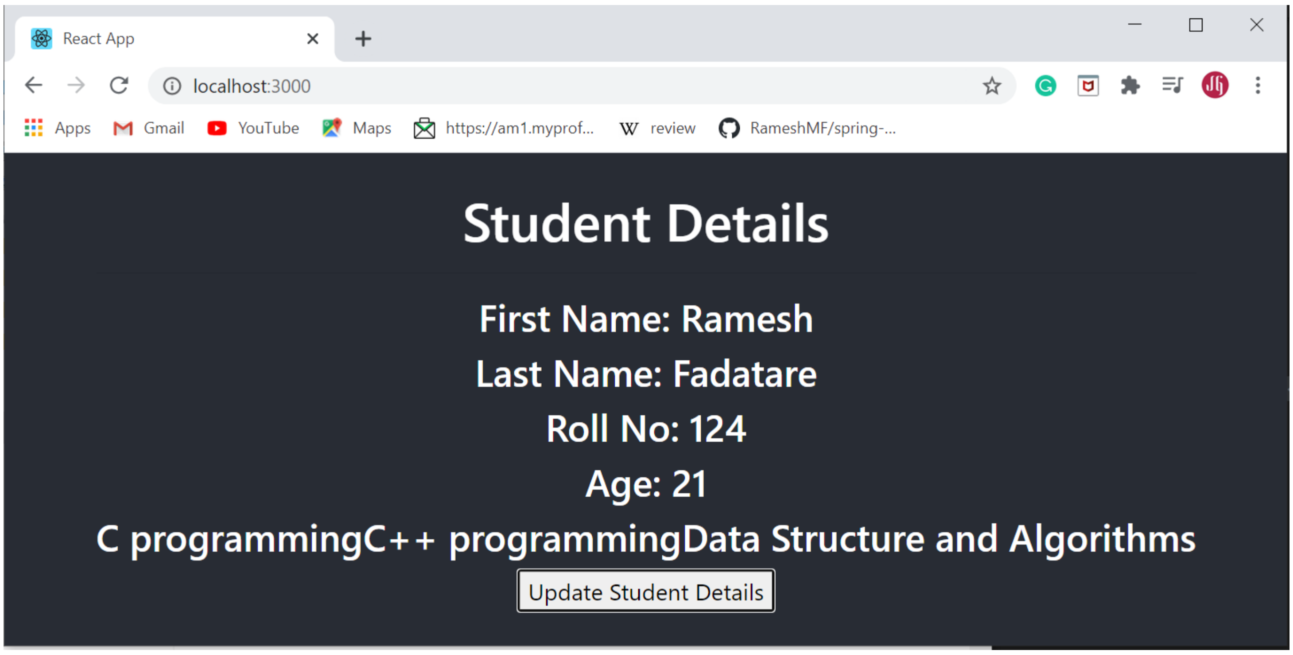
C programmingC++ programmingData Structure and Algorithms

Changing the state Object using setState() Method

To change a value in the state object, use `this.setState()` method.
When a value in the `state` object changes, the component will re-render, meaning that the output will change according to the new value(s).
For example, let's add a button with an `onClick` event that will change the **rollNo** and **age** of properties of the student:

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

class StudentComponent extends Component {
    constructor(props) {
        super(props)

        this.state = {
                 firstName: "Ramesh",
                 lastName:"Fadatare",
                 rollNo: 123,
                 age: 20,
                 books: ["C programming", "C++ programming",
"Data Structure and Algorithms"]
        }
    }

    updateStudent(){
        this.setState({
            rollNo: 124,
            age: 21
        })
    }
    render() {
        return (
            <div>
                <h1> Student Details</h1> <hr/>
                <h3> First Name: {this.state.firstName }</h3>
                <h3> Last Name: {this.state.lastName }</h3>
                <h3> Roll No: {this.state.rollNo } </h3>
                <h3> Age: {this.state.age }</h3>
                <h3> {this.state.books} </h3>
                <button type="button" onClick={() =>
this.updateStudent()} >Update Student Details</button>
            </div>
        )
    }
}

export default StudentComponent
```

Click on the "Update Student Details" button will update student details in the browser:



Student Details

First Name: Ramesh

Last Name: Fadatare

Roll No: 124

Age: 21

C programmingC++ programmingData Structure and Algorithms

What is Constructor?

The constructor is a method used to initialize an object's state in a class. It automatically called during the creation of an object in a class.

The concept of a constructor is the same in React. The constructor in a React component is called before the component is mounted. When you implement the constructor for a React component, you need to call **super(props)** method before any other statement. If you do not call super(props) method, **this.props** will be undefined in the constructor and can lead to bugs.

Syntax

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

In React, constructors are mainly used for two purposes:

It used for initializing the local state of the component by assigning an object to this.state.
It used for binding event handler methods that occur in your component.
Note: If you neither initialize state nor bind methods for your React component, there is no need to implement a constructor for React component.

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

Example

The concept of the constructor can understand from the below example.

Constructor1.js

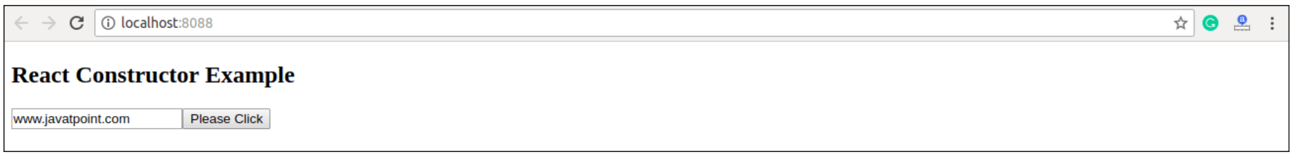
```
import React, { Component } from 'react';

class Constructor1 extends Component {
  constructor(props){
    super(props);
    this.state = {
        data: 'www.Imarticus.org'
      }
    this.handleEvent = this.handleEvent.bind(this);
  }
  handleEvent(){
    console.log(this.props);
  }
  render() {
    return (
      <div className="App">
    <h2>React Constructor Example</h2>
    <input type ="text" value={this.state.data} />
        <button onClick={this.handleEvent}>Please Click</button>
      </div>
    );
  }
}
export default Constructor1;
```

Main.js

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

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

Output

When you execute the above code, you get the following output.



localhost:8088

React Constructor Example

1. Is it necessary to have a constructor in every component?

No, it is not necessary to have a constructor in every component. If the component is not complex, it simply returns a node.

```
class App extends Component {
    render () {
        return (
            <p> Name: { this.props.name }</p>
        );
    }
}
```

2. Is it necessary to call super() inside a constructor?

Yes, it is necessary to call super() inside a constructor. If you need to set a property or access 'this' inside the constructor in your component, you need to call super().

```
class App extends Component {
    constructor(props){
        this.fName = "Jhon"; // 'this' is not allowed before super()
    }
    render () {
        return (
            <p> Name: { this.props.name }</p>
        );
    }
}
```

When you run the above code, you get an error saying **'this' is not allowed before super()**. So if you need to access the props inside the constructor, you need to call super(props).

Arrow Functions

The Arrow function is the new feature of the ES6 standard. If you need to use arrow functions, it is not necessary to bind any event to 'this.' Here, the scope of 'this' is global and not limited to any calling function. So If you are using Arrow Function, there is no need to bind 'this' inside the constructor.

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

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

We can use a constructor in the following ways:

1) The constructor is used to initialize state.

```
class App extends Component {
  constructor(props){
        // here, it is setting initial value for 'inputTextValue'
        this.state = {
            inputTextValue: 'initial value',
        };
    }
}
```
2) Using 'this' inside constructor

```
class App extends Component {
    constructor(props) {
        // when you use 'this' in constructor, super() needs to be called first
        super();
        // it means, when you want to use 'this.props' in constructor, call it as below
        super(props);
    }
}
```
3) Initializing third-party libraries

```
class App extends Component {
    constructor(props) {

        this.myBook = new MyBookLibrary();

        //Here, you can access props without using 'this'
        this.Book2 = new MyBookLibrary(props.environment);
    }
}
```
4) Binding some context(this) when you need a class method to be passed in props to children.

```
class App extends Component {
    constructor(props) {

        // when you need to 'bind' context to a function
        this.handleFunction = this.handleFunction.bind(this);
    }
}
```

Creating Form

React offers a stateful, reactive approach to build a form. The component rather than the DOM usually handles the React form. In React, the form is usually implemented by using controlled components.

There are mainly two types of form input in React.

Uncontrolled component
Controlled component
Uncontrolled component

The uncontrolled input is similar to the traditional HTML form inputs. The DOM itself handles the form data. Here, the HTML elements maintain their own state that will be updated when the input value changes. To write an uncontrolled component, you need to use a ref to get form values from the DOM. In other words, there is no need to write an event handler for every state update. You can use a ref to access the input field value of the form from the DOM.

Example

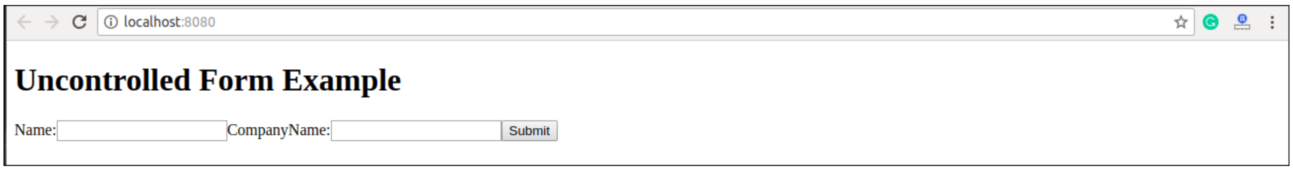
In this example, the code accepts a field **username** and **company name** in an uncontrolled component.

```
//Uncontrolled.js

import React, { Component } from 'react';
class Uncontrolled extends React.Component {
  constructor(props) {
      super(props);
      this.updateSubmit = this.updateSubmit.bind(this);
      this.input = React.createRef();
  }
  updateSubmit(event) {
      alert('You have entered the UserName and CompanyName successfully.');
      event.preventDefault();
  }
  render() {
    return (
      <form onSubmit={this.updateSubmit}>
        <h1>Uncontrolled Form Example</h1>
        <label>Name:
            <input type="text" ref={this.input} />
        </label>
        <label>
            CompanyName:
            <input type="text" ref={this.input} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

```
export default Uncontrolled;
```

Output

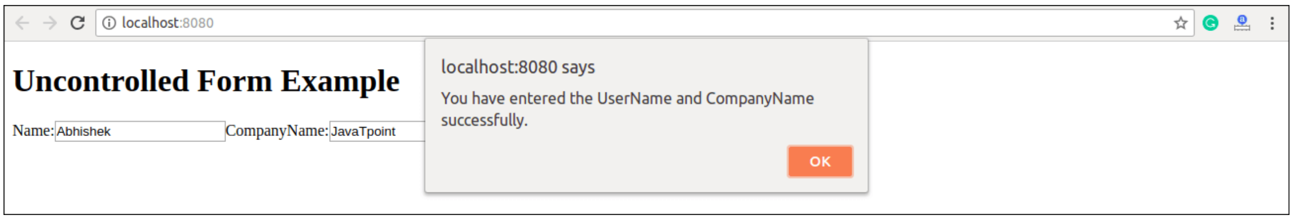
When you execute the above code, you will see the following screen.



Uncontrolled Form Example

Name: CompanyName: Submit

After filling the data in the field, you get the message that can be seen in the below screen.



Uncontrolled Form Example

Name: Abhishek CompanyName: JavaTpoint

localhost:8080 says

You have entered the UserName and CompanyName successfully.

Controlled Component

In HTML, form elements typically maintain their own state and update it according to the user input. In the controlled component, the input form element is handled by the component rather than the DOM. Here, the mutable state is kept in the state property and will be updated only with **setState()** method.

Controlled components have functions that govern the data passing into them on every **onChange event**, rather than grabbing the data only once, e.g., when you click a **submit button**. This data is then saved to state and updated with setState() method. This makes component have better control over the form elements and data.

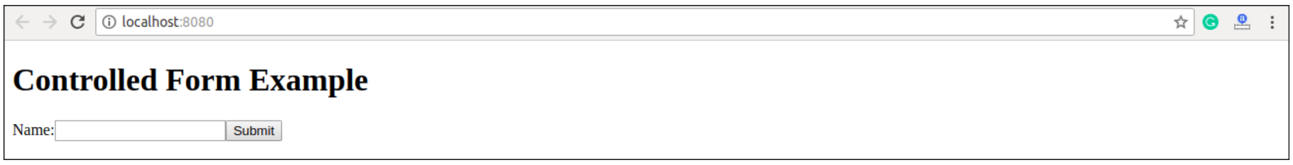
A controlled component takes its current value through **props** and notifies the changes through **callbacks** like an onChange event. A parent component "controls" this changes by handling the callback and managing its own state and then passing the new values as props to the controlled component. It is also called as a "dumb component."

Example

```
//Controlled.js

import React, { Component } from 'react';
class Controlled extends React.Component {
  constructor(props) {
     super(props);
     this.state = {value: ''};
     this.handleChange = this.handleChange.bind(this);
     this.handleSubmit = this.handleSubmit.bind(this);
  }
  handleChange(event) {
     this.setState({value: event.target.value});
  }
  handleSubmit(event) {
     alert('You have submitted the input successfully: ' + this.state.value);
     event.preventDefault();
  }
  render() {
     return (
         <form onSubmit={this.handleSubmit}>
           <h1>Controlled Form Example</h1>
           <label>
               Name:
                 <input type="text" value={this.state.value} onChange={this.handleChan
ge} />
           </label>
           <input type="submit" value="Submit" />
        </form>
     );
  }
}
export default Controlled;
```

Output

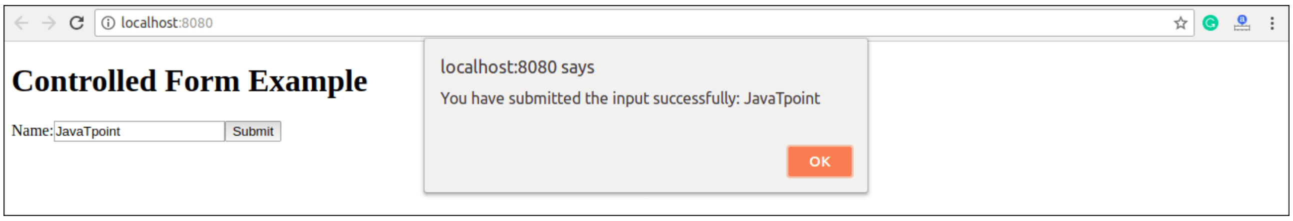
When you execute the above code, you will see the following screen.


localhost:8080

Controlled Form Example

Name: Submit

After filling the data in the field, you get the message that can be seen in the below screen.


localhost:8080

Controlled Form Example

Name: JavaTpoint Submit

localhost:8080 says

You have submitted the input successfully: JavaTpoint

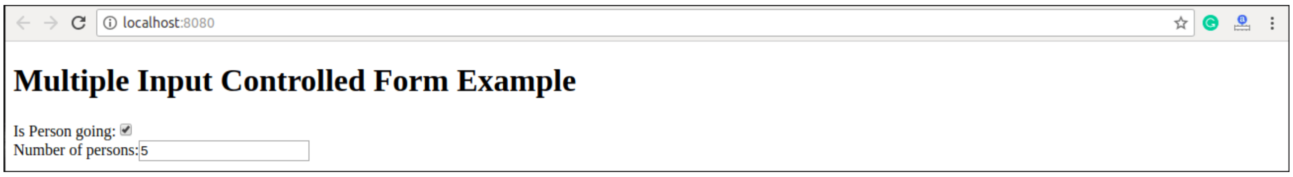
OK

Handling Multiple Inputs in Controlled Component

If you want to handle multiple controlled input elements, add a **name** attribute to each element, and then the handler function decided what to do based on the value of **event.target.name**.

Example

```
//MultipleInput.js

import React, { Component } from 'react';
class MultipleInput extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            personGoing: true,
            numberOfPersons: 5
        };
        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>
              <h1>Multiple Input Controlled Form Example</h1>
              <label>
                  Is Person going:
                  <input
                    name="personGoing"
                    type="checkbox"
                    checked={this.state.personGoing}
                    onChange={this.handleInputChange} />
             </label>
             <br />
             <label>
                 Number of persons:
                 <input
                 name="numberOfPersons"
                 type="number"
                 value={this.state.numberOfPersons}
                 onChange={this.handleInputChange} />
            </label>
        </form>
    );
  }
}
export default MultipleInput;
```
Output


localhost:8080

Multiple Input Controlled Form Example

Is Person going: ☑
Number of persons: 5

React Conditional Rendering

In React, we can create multiple components which encapsulate behavior that we need. After that, we can render them depending on some conditions or the state of our application. In other words, based on one or several conditions, a component decides which elements it will return. In React, conditional rendering works the same way as the conditions work in JavaScript. We use JavaScript operators to create elements representing the current state, and then React Component update the UI to match them.

From the given scenario, we can understand how conditional rendering works. Consider an example of handling a **login/logout** button. The login and logout buttons will be separate components. If a user logged in, render the **logout component** to display the logout button. If a user not logged in, render the **login component** to display the login button. In React, this situation is called as **conditional rendering**.

There is more than one way to do conditional rendering in React. They are given below.

if
ternary operator
logical && operator
switch case operator
Conditional Rendering with enums

if

It is the easiest way to have a conditional rendering in React in the render method. It is restricted to the total block of the component. IF the condition is **true**, it will return the element to be rendered. It can be understood in the below example.

Example

```
function UserLoggin(props) {
  return <h1>Welcome back!</h1>;
}
function GuestLoggin(props) {
  return <h1>Please sign up.</h1>;
}
function SignUp(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserLogin />;
  }
  return <GuestLogin />;
}

ReactDOM.render(
  <SignUp isLoggedIn={false} />,
  document.getElementById('root')
);
```

Logical && operator

This operator is used for checking the condition. If the condition is **true**, it will return the element **right** after **&&**, and if it is **false**, React will **ignore** and skip it.

Syntax

```
{
    condition &&
    // whatever written after && will be a part of output.
}
```

We can understand the behavior of this concept from the below example.

If you run the below code, you will not see the **alert** message because the condition is not matching.

```
('Imarticus' == 'Imarticus') && alert('This alert will never be shown!')
```

If you run the below code, you will see the **alert** message because the condition is matching.

```
(10 > 5) && alert('This alert will be shown!')
```

Example

```
import React from 'react';
import ReactDOM from 'react-dom';
// Example Component
function Example()
```

```
{
   return(<div>
          {
             (10 > 5) && alert('This alert will be shown!')
          }
         </div>
       );
}
```

You can see in the above output that as the condition **(10 > 5)** evaluates to true, the alert message is successfully rendered on the screen.

Ternary operator

The ternary operator is used in cases where two blocks alternate given a certain condition. This operator makes your if-else statement more concise. It takes **three** operands and used as a shortcut for the if statement.

Syntax

```
condition ?  true : false
```

If the condition is **true**, **statement1** will be rendered. Otherwise, **false** will be rendered.

Example

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      Welcome {isLoggedIn ? 'Back' : 'Please login first'}.
    </div>
  );
}
```

Switch case operator

Sometimes it is possible to have multiple conditional renderings. In the switch case, conditional rendering is applied based on a different state.

Example

```
function NotificationMsg({ text}) {
  switch(text) {
    case 'Hi All':
      return <Message: text={text} />;
    case 'Hello Imarticus':
      return <Message text={text} />;
    default:
      return null;
  }
}
```

if-else

In your code editor, create a new `AuthButton.js` file:

```
import React from "react";

const AuthButton = props => {
  let { isLoggedIn } = props;

  if (isLoggedIn) {
    return <button>Logout</button>;
  } else {
    return <button>Login</button>;
  }
};

export default AuthButton;
```

Next, revisit `App.js` and modify it to use the new component:

```
import React, { Component } from "react";
import './App.css';
import AuthButton from "./AuthButton";

class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      isLoggedIn: true
    };
  }

  render() {
    return (
      <div className="App">
        <h1>
          This is a Demo showing several ways to implement Conditional Rendering in
React.
        </h1>
        <AuthButton isLoggedIn={isLoggedIn} />
      </div>
    );
  }
}

export default App;
```

2. Using a `switch` Statement

As shown previously, you can conditionally return different markup from a component based on set conditions using an `if…else` statement. The same could be achieved with a `switch` statement where you can specify the markup for various conditions.

Revisi the `AuthButton` component and replace the `if…else` statement with a `switch` statement:

```
import React from "react";

const AuthButton = props => {
  let { isLoggedIn } = props;

  switch (isLoggedIn) {
    case true:
      return <button>Logout</button>;
      break;
    case false:
      return <button>Login</button>;
      break;
    default:
      return null;
  }
};

export default AuthButton;
```

3. Using Element Variables

Element variables are similar to the approach to extract the conditional rendering into a function. Element variables are variables that hold JSX elements. You can conditionally assign elements or components to these variables outside the JSX and only render the variable within JSX.

```
import React, { Component } from "react";
import './App.css';

class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      isLoggedIn: true
    };
  }

  render() {
    let { isLoggedIn } = this.state;
    let AuthButton;

    if (isLoggedIn) {
      AuthButton = <button>Logout</button>;
    } else {
      AuthButton = <button>Login</button>;
    }

    return (
      <div className="App">
        <h1>
          This is a Demo showing several ways to implement Conditional Rendering in
React.
        </h1>
        {AuthButton}
      </div>
    );
  }
}

export default App;
```

4. Using Ternary Operators

The conditional (ternary) operator is the only JavaScript operator that takes three operands. This operator is frequently used as a shortcut for the `if` statement.

```
import React, { Component } from "react";
import './App.css';

class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      isLoggedIn: true
    };
  }

  render() {
    let { isLoggedIn } = this.state;

    return (
      <div className="App">
        <h1>
          This is a Demo showing several ways to implement Conditional Rendering in
React.
        </h1>
        {isLoggedIn ? <button>Logout</button> : <button>Login</button>}
      </div>
    );
  }
}

export default App;
```

5. Using Logical && (Short Circuit Evaluation)

Short circuit evaluation is a technique used to ensure that there are no side effects during the evaluation of operands in an expression. The logical && helps you specify that an action should be taken only on one condition, otherwise, it would be ignored entirely.

```
import React, { Component } from "react";
import './App.css';

class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      isLoggedIn: true
    };
  }

  render() {
    let { isLoggedIn } = this.state;

    return (
      <div className="App">
```

```
        <h1>
          This is a Demo showing several ways to implement Conditional Rendering in
React.
        </h1>
        {isLoggedIn && <button>Logout</button>}
      </div>
    );
  }
}

export default App;
```

6. Using Immediately Invoked Function Expressions (IIFEs)

Earlier sections mentioned that JSX limitations make it unable to execute every type of JavaScript code. It is possible to bypass these limitations with Immediately Invoked Function Expressions (IFFEs). IFFEs is a JavaScript function that runs as soon as it is defined:

```
import React, { Component } from "react";
import './App.css';

class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      isLoggedIn: true
    };
  }

  render() {
    let { isLoggedIn } = this.state;

    return (
      <div className="App">
        <h1>
          This is a Demo showing several ways to implement Conditional Rendering in
React.
        </h1>
        {(function() {
          if (isLoggedIn) {
            return <button>Logout</button>;
          } else {
            return <button>Login</button>;
          }
      })()}
    </div>
  );
 }
}

export default App;
```

Conditional Rendering with enums

An **enum** is a great way to have a multiple conditional rendering. It is more **readable** as compared to switch case operator. It is perfect for **mapping** between different **state**. It is also perfect for mapping in more than one condition. It can be understood in the below example.

Example

```
function NotificationMsg({ text, state }) {
 return (
  <div>
   {{
    info: <Message text={text} />,
    warning: <Message text={text} />,
   }[state]}
  </div>
 );
}
```

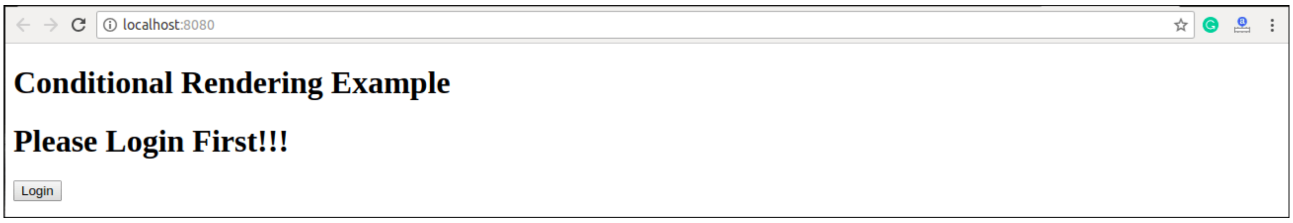
Conditional Rendering Example

In the below example, we have created a **stateful** component called **App** which maintains the login control. Here, we create three components representing Logout, Login, and Message component. The stateful component App will render either or depending on its current **state**.

```
import React, { Component } from 'react';
// Message Component
function Message(props)
{
  if (props.isLoggedIn)
    return <h1>Welcome Back!!!</h1>;
  else
    return <h1>Please Login First!!!</h1>;
}
// Login Component
```

```
function Login(props)
{
  return(
          <button onClick = {props.clickInfo}> Login </button>
      );
}
// Logout Component
function Logout(props)
{
   return(
          <button onClick = {props.clickInfo}> Logout </button>
      );
}
class App extends Component{
  constructor(props)
   {
      super(props);
      this.handleLogin = this.handleLogin.bind(this);
        this.handleLogout = this.handleLogout.bind(this);
     this.state = {isLoggedIn : false};
   }
  handleLogin()
   {
      this.setState({isLoggedIn : true});
   }
   handleLogout()
   {
      this.setState({isLoggedIn : false});
   }
   render(){
      return(
          <div>
      <h1> Conditional Rendering Example </h1>
              <Message isLoggedIn = {this.state.isLoggedIn}/>
              {
                  (this.state.isLoggedIn)?(
                  <Logout clickInfo = {this.handleLogout} />
                  ) : (
                  <Login clickInfo = {this.handleLogin} />
                  )
              }
          </div>
          );
   }
}
export default App;
```
Output:

When you execute the above code, you will get the following screen.

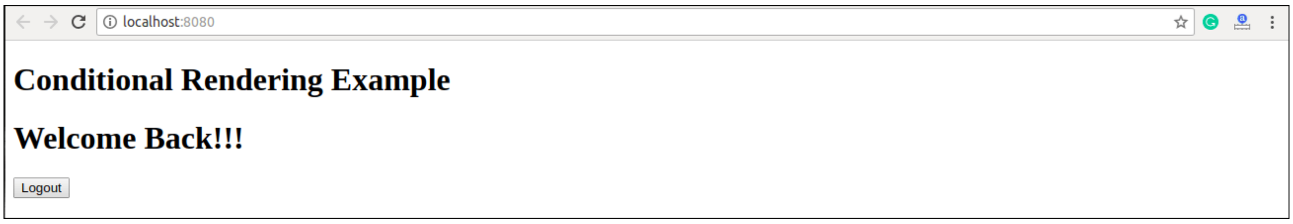


localhost:8080

Conditional Rendering Example

Please Login First!!!

After clicking the logout button, you will get the below screen.



localhost:8080

Conditional Rendering Example

Welcome Back!!!

Preventing Component form Rendering

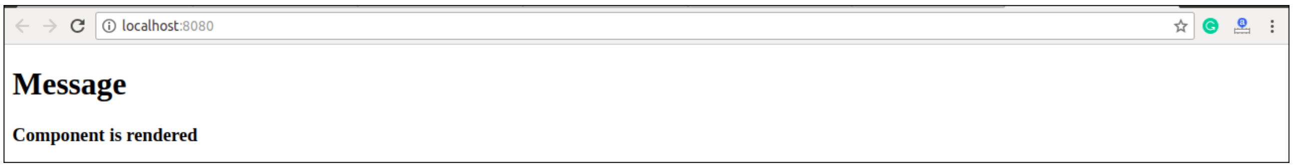
Sometimes it might happen that a component hides itself even though another component rendered it. To do this (prevent a component from rendering), we will have to return **null** instead of its render output. It can be understood in the below example:

Example

In this example, the is rendered based on the value of the prop called **displayMessage**. If the prop value is false, then the component does not render.

```
import React from 'react';
import ReactDOM from 'react-dom';
function Show(props)
{
    if(!props.displayMessage)
        return null;
    else
        return <h3>Component is rendered</h3>;
}
ReactDOM.render(
    <div>
        <h1>Message</h1>
        <Show displayMessage = {true} />
    </div>,
    document.getElementById('app')
```

);
Output:


localhost:8080

Message

Component is rendered

React Lists

Lists are used to display data in an ordered format and mainly used to display menus on websites. In React, Lists can be created in a similar way as we create lists in JavaScript. Let us see how we transform Lists in regular JavaScript.

The map() function is used for traversing the lists. In the below example, the map() function takes an array of numbers and multiply their values with 5. We assign the new array returned by map() to the variable multiplyNums and log it.

Example

```
var numbers = [1, 2, 3, 4, 5];
const multiplyNums = numbers.map((number)=>{
   return (number * 5);
});
console.log(multiplyNums);
```
Output

The above JavaScript code will log the output on the console. The output of the code is given below.

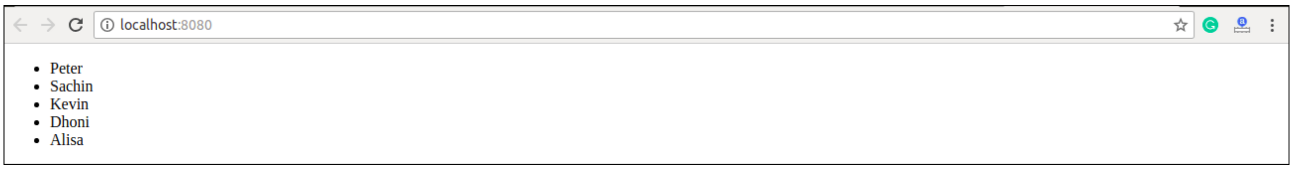
```
[5, 10, 15, 20, 25]
```

Now, let us see how we create a list in React. To do this, we will use the map() function for traversing the list element, and for updates, we enclosed them between **curly braces {}**. Finally, we assign the array elements to listItems. Now, include this new list inside **<ul> </ul>** elements and render it to the DOM.

Example

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

const myList = ['Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa'];
const listItems = myList.map((myList)=>{
   return <li>{myList}</li>;
});
```

```
ReactDOM.render(
    <ul> {listItems} </ul>,
    document.getElementById('app')
);
export default App;
```
Output



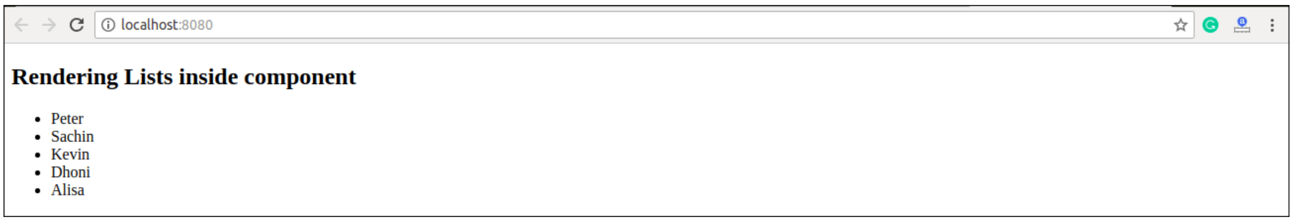
```
• Peter
• Sachin
• Kevin
• Dhoni
• Alisa
```

Rendering Lists inside components

In the previous example, we had directly rendered the list to the DOM. But it is not a good practice to render lists in React. In React, we had already seen that everything is built as individual components. Hence, we would need to render lists inside a component. We can understand it in the following code.

Example

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

function NameList(props) {
  const myLists = props.myLists;
  const listItems = myLists.map((myList) =>
    <li>{myList}</li>
  );
  return (
    <div>
        <h2>Rendering Lists inside component</h2>
              <ul>{listItems}</ul>
    </div>
  );
}
const myLists = ['Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa'];
ReactDOM.render(
  <NameList myLists={myLists} />,
  document.getElementById('app')
);
export default App;
```
Output


localhost:8080

Rendering Lists inside component

- Peter
- Sachin
- Kevin
- Dhoni
- Alisa

React Keys

A key is a unique identifier. In React, it is used to identify which items have changed, updated, or deleted from the Lists. It is useful when we dynamically created components or when the users alter the lists. It also helps to determine which components in a collection needs to be re-rendered instead of re-rendering the entire set of components every time.

Keys should be given inside the array to give the elements a stable identity. The best way to pick a key as a string that uniquely identifies the items in the list. It can be understood with the below example.

Example

```
const stringLists = [ 'Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa' ];

const updatedLists = stringLists.map((strList)=>{
    <li key={strList.id}> {strList} </li>;
});
```

If there are no stable IDs for rendered items, you can assign the item **index** as a key to the lists. It can be shown in the below example.

Example

```
const stringLists = [ 'Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa' ];

const updatedLists = stringLists.map((strList, index)=>{
    <li key={index}> {strList} </li>;
});
```

Note: It is not recommended to use indexes for keys if the order of the item may change in future. It creates confusion for the developer and may cause issues with the component state.

Using Keys with component

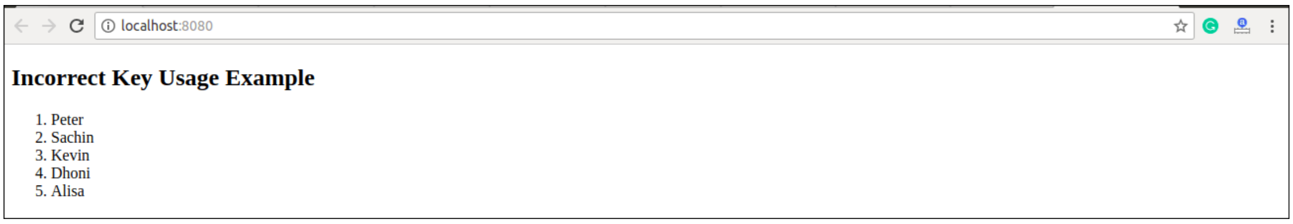
Consider you have created a separate component for **ListItem** and extracting ListItem from that component. In this case, you should have to assign keys on the **<ListItem />** elements in the array, not to the **<li>** elements in the ListItem itself. To avoid mistakes, you have to keep in mind that keys only make sense in the context of the surrounding array. So, anything you are returning from map() function is recommended to be assigned a key.

Example: Incorrect Key usage

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

function ListItem(props) {
  const item = props.item;
  return (
    // Wrong! No need to specify the key here.
    <li key={item.toString()}>
      {item}
    </li>
  );
}
function NameList(props) {
  const myLists = props.myLists;
  const listItems = myLists.map((strLists) =>
    // The key should have been specified here.
    <ListItem item={strLists} />
  );
  return (
    <div>
        <h2>Incorrect Key Usage Example</h2>
              <ol>{listItems}</ol>
    </div>
  );
}
const myLists = ['Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa'];
ReactDOM.render(
  <NameList myLists={myLists}/>,
  document.getElementById('app')
);
export default App;
```

In the given example, the list is rendered successfully. But it is not a good practice that we had not assigned a key to the map() iterator.

Output


localhost:8080

Incorrect Key Usage Example

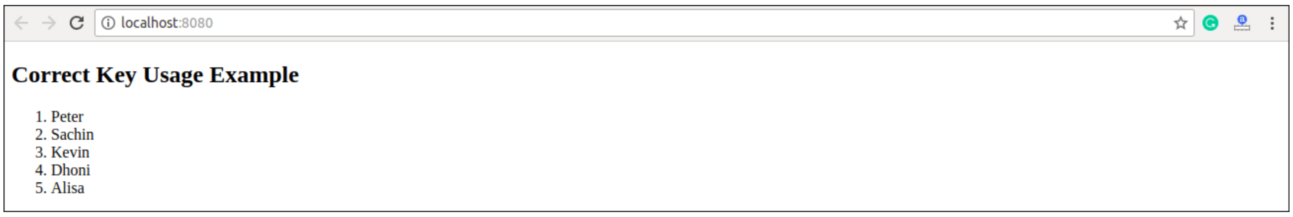
1. Peter
2. Sachin
3. Kevin
4. Dhoni
5. Alisa

Example: Correct Key usage

To correct the above example, we should have to assign key to the map() iterator.

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

function ListItem(props) {
  const item = props.item;
  return (
    // No need to specify the key here.
    <li> {item} </li>
  );
}
function NameList(props) {
  const myLists = props.myLists;
  const listItems = myLists.map((strLists) =>
    // The key should have been specified here.
    <ListItem key={myLists.toString()} item={strLists} />
  );
  return (
    <div>
        <h2>Correct Key Usage Example</h2>
            <ol>{listItems}</ol>
    </div>
  );
}
const myLists = ['Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa'];
ReactDOM.render(
  <NameList myLists={myLists}/>,
  document.getElementById('app')
);
export default App;
```

Output


localhost:8080

Correct Key Usage Example

1. Peter
2. Sachin
3. Kevin
4. Dhoni
5. Alisa

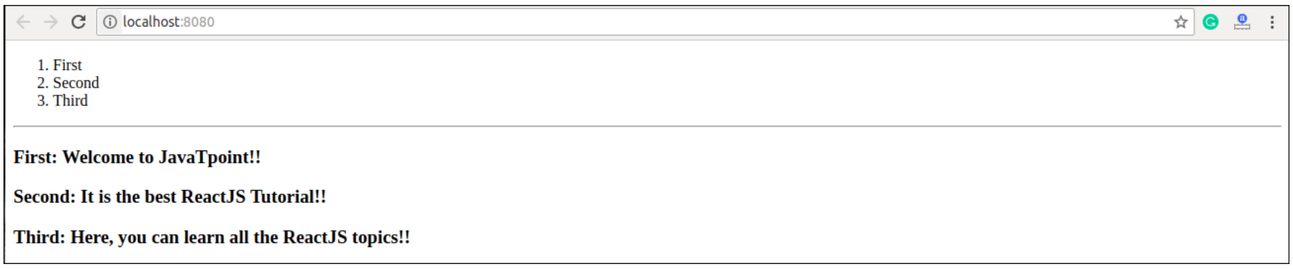
Uniqueness of Keys among Siblings

We had discussed that keys assignment in arrays must be unique among their **siblings**. However, it doesn't mean that the keys should be **globally** unique. We can use the same set of keys in producing two different arrays. It can be understood in the below example.

Example

```
import React from 'react';
import ReactDOM from 'react-dom';
function MenuBlog(props) {
  const titlebar = (
    <ol>
      {props.data.map((show) =>
        <li key={show.id}>
          {show.title}
        </li>
      )}
    </ol>
  );
  const content = props.data.map((show) =>
    <div key={show.id}>
      <h3>{show.title}: {show.content}</h3>
    </div>
  );
  return (
    <div>
      {titlebar}
      <hr />
      {content}
    </div>
  );
}
const data = [
  {id: 1, title: 'First', content: 'Welcome to Imarticus!!'},
  {id: 2, title: 'Second', content: 'It is the best ReactJS Tutorial!!'},
  {id: 3, title: 'Third', content: 'Here, you can learn all the ReactJS topics!!'}
];
ReactDOM.render(
  <MenuBlog data={data} />,
  document.getElementById('app')
);
export default App;
```

Output



```
localhost:8080

1. First
2. Second
3. Third

First: Welcome to JavaTpoint!!

Second: It is the best ReactJS Tutorial!!

Third: Here, you can learn all the ReactJS topics!!
```

React Refs

Refs is the shorthand used for **references** in React. It is similar to **keys** in React. It is an attribute which makes it possible to store a reference to particular DOM nodes or React elements. It provides a way to access React DOM nodes or React elements and how to interact with it. It is used when we want to change the value of a child component, without making the use of props.

When to Use Refs

Refs can be used in the following cases:

When we need DOM measurements such as managing focus, text selection, or media playback.
It is used in triggering imperative animations.
When integrating with third-party DOM libraries.
It can also use as in callbacks.

When to not use Refs

Its use should be avoided for anything that can be done **declaratively**. For example, instead of using **open()** and **close()** methods on a Dialog component, you need to pass an **isOpen** prop to it.
You should have to avoid overuse of the Refs.

How to create Refs

In React, Refs can be created by using **React.createRef()**. It can be assigned to React elements via the **ref** attribute. It is commonly assigned to an instance property when a component is created, and then can be referenced throughout the component.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.callRef = React.createRef();
  }
  render() {
    return <div ref={this.callRef} />;
  }
}
```

How to access Refs

In React, when a ref is passed to an element inside render method, a reference to the node can be accessed via the current attribute of the ref.

```
const node = this.callRef.current;
```

Refs current Properties

The ref value differs depending on the type of the node:

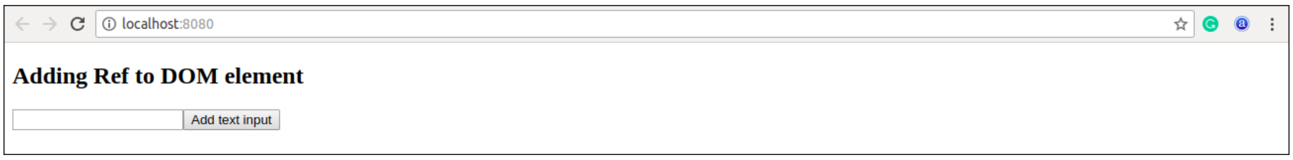
When the ref attribute is used in HTML element, the ref created with **React.createRef()** receives the underlying DOM element as its **current** property.
If the ref attribute is used on a custom class component, then ref object receives the **mounted** instance of the component as its current property.
The ref attribute cannot be used on **function components** because they don't have instances.

Add Ref to DOM elements

In the below example, we are adding a ref to store the reference to a DOM node or element.

```
import React, { Component } from 'react';
import { render } from 'react-dom';

class App extends React.Component {
  constructor(props) {
    super(props);
    this.callRef = React.createRef();
    this.addingRefInput = this.addingRefInput.bind(this);
  }

  addingRefInput() {
    this.callRef.current.focus();
  }

  render() {
    return (
      <div>
        <h2>Adding Ref to DOM element</h2>
        <input
          type="text"
          ref={this.callRef} />
        <input
          type="button"
          value="Add text input"
          onClick={this.addingRefInput}
        />
      </div>
    );
  }
}
export default App;
```

Output


localhost:8080

Adding Ref to DOM element

Add Ref to Class components

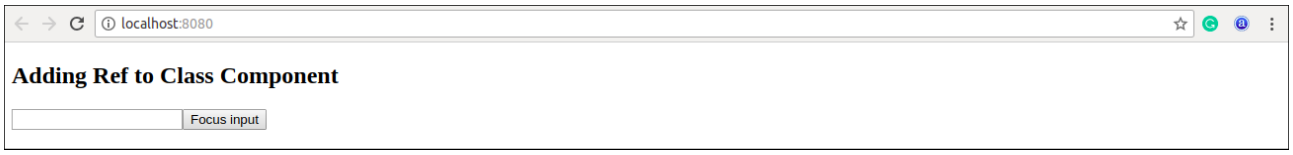
In the below example, we are adding a ref to store the reference to a class component.

Example

```
import React, { Component } from 'react';
import { render } from 'react-dom';

function CustomInput(props) {
  let callRefInput = React.createRef();

  function handleClick() {
    callRefInput.current.focus();
  }

  return (
    <div>
      <h2>Adding Ref to Class Component</h2>
      <input
        type="text"
        ref={callRefInput} />
      <input
```

```
        type="button"
        value="Focus input"
        onClick={handleClick}
      />
    </div>
  );
}
class App extends React.Component {
  constructor(props) {
    super(props);
    this.callRefInput = React.createRef();
  }

  focusRefInput() {
    this.callRefInput.current.focus();
  }

  render() {
    return (
      <CustomInput ref={this.callRefInput} />
    );
  }
}
export default App;
```

Output


localhost:8080

Adding Ref to Class Component

Callback refs

In react, there is another way to use refs that is called "**callback refs**" and it gives more control when the refs are **set** and **unset**. Instead of creating refs by createRef() method, React allows a way to create refs by passing a callback function to the ref attribute of a component. It looks like the below code.

```
<input type="text" ref={element => this.callRefInput = element} />
```

The callback function is used to store a reference to the DOM node in an instance property and can be accessed elsewhere. It can be accessed as below:

```
this.callRefInput.value
```

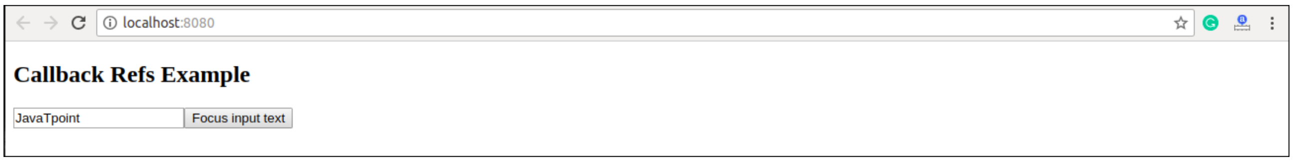
The example below helps to understand the working of callback refs.

```
import React, { Component } from 'react';
import { render } from 'react-dom';
```

```
class App extends React.Component {
    constructor(props) {
    super(props);

    this.callRefInput = null;

    this.setInputRef = element => {
      this.callRefInput = element;
    };

    this.focusRefInput = () => {
      //Focus the input using the raw DOM API
      if (this.callRefInput) this.callRefInput.focus();
    };
  }

  componentDidMount() {
    //autofocus of the input on mount
    this.focusRefInput();
  }

  render() {
    return (
      <div>
    <h2>Callback Refs Example</h2>
        <input
          type="text"
          ref={this.setInputRef}
        />
        <input
          type="button"
          value="Focus input text"
          onClick={this.focusRefInput}
        />
      </div>
    );
  }
}
export default App;
```

In the above example, React will call the "ref" callback to store the reference to the input DOM element when the component **mounts**, and when the component **unmounts**, call it with **null**. Refs are always **up-to-date** before the **componentDidMount** or **componentDidUpdate** fires. The callback refs pass between components is the same as you can work with object refs, which is created with React.createRef().

Output


localhost:8080

Callback Refs Example

JavaTpoint | Focus input text

Forwarding Ref from one component to another component

Ref forwarding is a technique that is used for passing a **ref** through a component to one of its child components. It can be performed by making use of the **React.forwardRef()** method. This technique is particularly useful with **higher-order components** and specially used in reusable component libraries. The most common example is given below.

Example

```
import React from 'react';

const TextInput = React.forwardRef((props, ref) => (
  <input type="text" placeholder="Hello World" ref={ref} />
));

const inputRef = React.createRef();

class CustomTextInput extends React.Component {
  handleSubmit = e => {
    e.preventDefault();
    console.log(inputRef.current.value);
  };
  render() {
    return (
      <div>
        <form onSubmit={e => this.handleSubmit(e)}>
          <TextInput ref={inputRef} />
          <button>Submit</button>
        </form>
      </div>
    );
  }
}
export default CustomTextInput;
```

In the above example, there is a component **TextInput** that has a child as an input field. Now, to pass or forward the **ref** down to the input, first, create a ref and then pass your ref down to **<TextInput ref={inputRef}>**. After that, React forwards the ref to the **forwardRef** function as a second argument. Next, we forward this ref argument down to **<input ref={ref}>**. Now, the value of the DOM node can be accessed at **inputRef.current**.

React with useRef()

It is introduced in **React 16.7** and above version. It helps to get access the DOM node or element, and then we can interact with that DOM node or element such as focussing the input element or accessing the input element value. It returns the ref object whose **.current** property initialized to the passed argument. The returned object persist for the lifetime of the component.

Syntax

```
const refContainer = useRef(initialValue);
```

Example

In the below code, **useRef** is a function that gets assigned to a variable, **inputRef**, and then attached to an attribute called ref inside the HTML element in which you want to reference.

```
function useRefExample() {
  const inputRef= useRef(null);
  const onButtonClick = () => {
    inputRef.current.focus();
  };
  return (
    <>
      <input ref={inputRef} type="text" />
      <button onClick={onButtonClick}>Submit</button>
    </>
  );
}
```

React Fragments

In React, whenever you want to render something on the screen, you need to use a render method inside the component. This render method can return **single** elements or **multiple** elements. The render method will only render a single root node inside it at a time. However, if you want to return multiple elements, the render method will require a '**div**' tag and put the entire content or elements inside it.

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

Syntax

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

Example

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

Why we use Fragments?

The main reason to use Fragments tag is:

It makes the execution of code faster as compared to the div tag.
It takes less memory.

Fragments Short Syntax

There is also another shorthand exists for declaring fragments for the above method. It looks like **empty** tag in which we can use of '<>' and '' instead of the '**React.Fragment**'.

Example

```
//Rendering with short syntax
class Columns extends React.Component {
  render() {
    return (
      <>
        <h2> Hello World! </h2>
        <p> Welcome to the Imarticus </p>
      </>
    );
  }
}
```

Keyed Fragments

The shorthand syntax does not accept key attributes. You need a key for mapping a collection to an array of fragments such as to create a description list. If you need to provide keys, you have to declare the fragments with the explicit <**React.Fragment**> syntax.

Note: Key is the only attributes that can be passed with the Fragments.

Example

```
Function  = (props) {
  return (
    <Fragment>
      {props.items.data.map(item => (
        // Without the 'key', React will give a key warning
        <React.Fragment key={item.id}>
          <h2>{item.name}</h2>
          <p>{item.url}</p>
          <p>{item.description}</p>
        </React.Fragment>
      ))}
    </Fragment>
  )
}
```

Need of React Router

React Router plays an important role to display multiple views in a single page application. Without React Router, it is not possible to display multiple views in React applications. Most of the social media websites like Facebook, Instagram uses React Router for rendering multiple views.

React Router Installation

React contains three different packages for routing. These are:

react-router: It provides the core routing components and functions for the React Router applications.
react-router-native: It is used for mobile applications.
react-router-dom: It is used for web applications design.
It is not possible to install react-router directly in your application. To use react routing, first, you need to install react-router-dom modules in your application. The below command is used to install react router dom.

```
$ npm install react-router-dom --save
```

Components in React Router

There are two types of router components:

<BrowserRouter>: It is used for handling the dynamic URL.
<HashRouter>: It is used for handling the static request.
Example

Step-1: In our project, we will create two more components along with **App.js**, which is already present.

About.js

```
import React from 'react'
class About extends React.Component {
  render() {
    return <h1>About</h1>
  }
}
export default About
```

Contact.js

```
import React from 'react'
class Contact extends React.Component {
  render() {
    return <h1>Contact</h1>
  }
}
export default Contact
```

App.js

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

```
class App extends React.Component {
  render() {
    return (
      <div>
        <h1>Home</h1>
      </div>
    )
  }
}
export default App
```

Step-2: For Routing, open the index.js file and import all the three component files in it. Here, you need to import line: **import { Route, Link, BrowserRouter as Router } from 'react-router-dom'** which helps us to implement the Routing. Now, our index.js file looks like below.

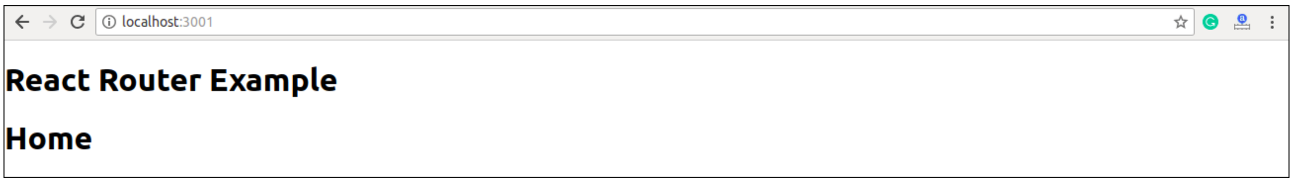
What is Route?

It is used to define and render component based on the specified path. It will accept components and render to define what should be rendered.

Index.js

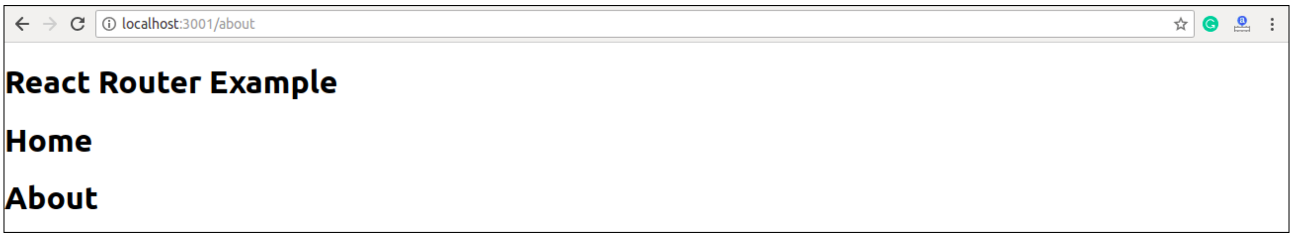
```
import React from 'react';
import ReactDOM from 'react-dom';
import {BrowserRouter, Route, Routes } from 'react-router-
dom';

import './index.css';
import App from './App';
import About from './about'
import Contact from './contact'

const routing = (

  <div>
    <div>
       <Routes>
         <Route path="/about" element={<About />} />
         <Route path="/contact" element={<Contact />} />
         <Route path="/location" element={<Place />} />
       </Routes>
     </div>
  </div>

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

Step-3: Open **command prompt**, go to your project location, and then type **npm start**. You will get the following screen.



```
React Router Example

Home
```

Now, if you enter **manually** in the browser: **localhost:3000/about**, you will see **About** component is rendered on the screen.



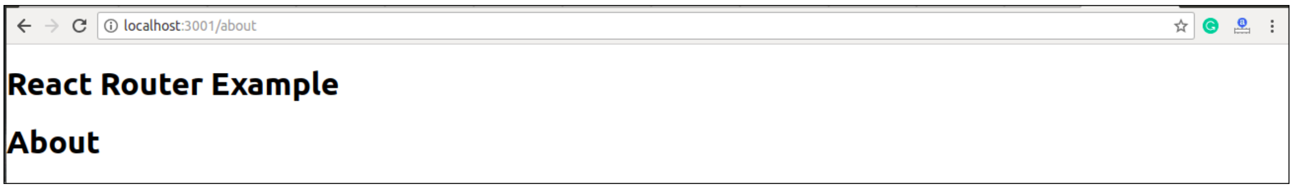
```
React Router Example

Home

About
```

Step-4: In the above screen, you can see that **Home** component is still rendered. It is because the home path is '**/**' and about path is '**/about**', so you can observe that **slash** is common in both paths which render both components. To stop this behavior, you need to use the **exact** prop. It can be seen in the below example.

Index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Route, Link, BrowserRouter as Router } from 'react-router-dom'
import './index.css';
import App from './App';
import About from './about'
import Contact from './contact'

const routing = (
  <Router>
    <div>
      <h1>React Router Example</h1>
      <Route exact path="/" component={App} />
      <Route path="/about" component={About} />
      <Route path="/contact" component={Contact} />
    </div>
  </Router>
)
ReactDOM.render(routing, document.getElementById('root'));
```

Output


localhost:3001/about

React Router Example

About

Adding Navigation using Link component

Sometimes, we want to need **multiple** links on a single page. When we click on any of that particular **Link**, it should load that page which is associated with that path without **reloading** the web page. To do this, we need to import **<Link>** component in the **index.js** file.

What is < Link> component?

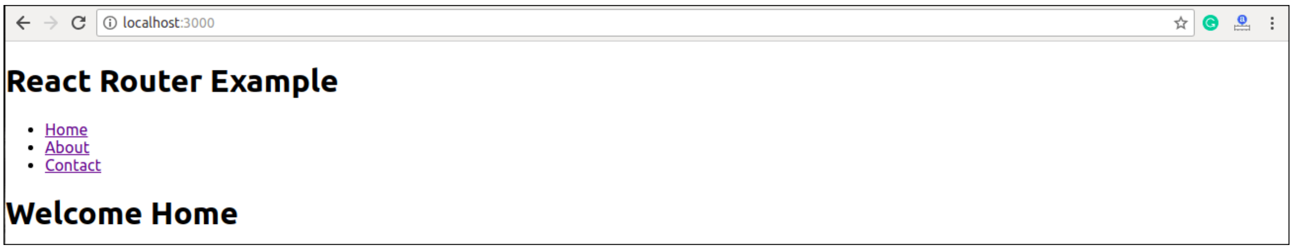
This component is used to create links which allow to **navigate** on different **URLs** and render its content without reloading the webpage.

Example

Index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Route, Link, BrowserRouter as Router } from 'react-router-dom'
import './index.css';
import App from './App';
import About from './about'
import Contact from './contact'

const routing = (
  <Router>
    <div>
      <h1>React Router Example</h1>
      <ul>
        <li>
          <Link to="/">Home</Link>
        </li>
        <li>
          <Link to="/about">About</Link>
        </li>
        <li>
          <Link to="/contact">Contact</Link>
        </li>
      </ul>
      <Route exact path="/" component={App} />
      <Route path="/about" component={About} />
      <Route path="/contact" component={Contact} />
    </div>
  </Router>
)
ReactDOM.render(routing, document.getElementById('root'));
```

Output



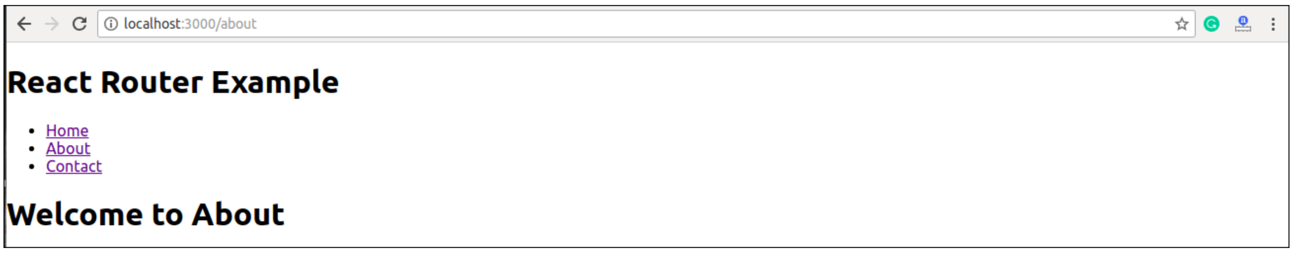
```
localhost:3000
```

React Router Example

- Home
- About
- Contact

Welcome Home

After adding Link, you can see that the routes are rendered on the screen. Now, if you click on the **About**, you will see URL is changing and About component is rendered.



```
localhost:3000/about
```

React Router Example

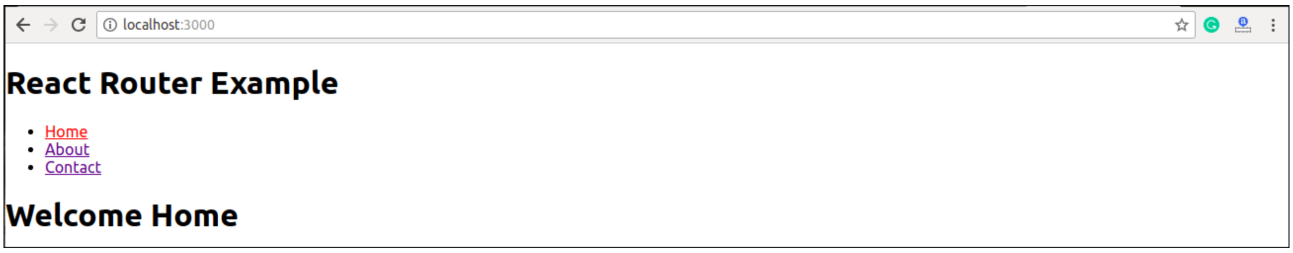
- Home
- About
- Contact

Welcome to About

```
import './App.css';
import {BrowserRouter, Route, Routes,Link } from 'react-
router-dom';
import Test from './Test';
import Welcome from './Welcome';
import About from './About';
import Place from './Location';
import Contact from './Contact'
import Addition from './Additional';
import Array from './Array';
import Login from './Login';
import Employee from './Employee';
import Counter from './Counter';
import Greeting from './Greeting';
import Welcome1 from './Welcome1';
import { Student } from './Student';
import { Student1 } from './PropsDemo';
import { Skills } from './Skills';
import Company from './Company';
import LoginForm from './LoginForm';
import Constructor1 from './Constructor1';
import Example from './Example';
import NotificationMsg from './NotificationMsg';

function App() {
  //const skills = ['HTML', 'CSS', 'JavaScript']

  return (
    <div>
      <div>
      <nav>
            <Link to ="/"> Home </Link> ||
            <Link to ="/contact"> Contact </Link> ||
            <Link to ="/location"> Location </Link>
       </nav>
      <Routes>
        <Route path="/" element={<About />} />
        <Route path="/contact" element={<Contact />} />
        <Route path="/location" element={<Place />} />
      </Routes>
    </div>

    </div>
  );
}

export default App;
```

Output

When we execute the above program, we will get the following screen in which we can see that **Home** link is of color **Red** and is the only currently **active** link.



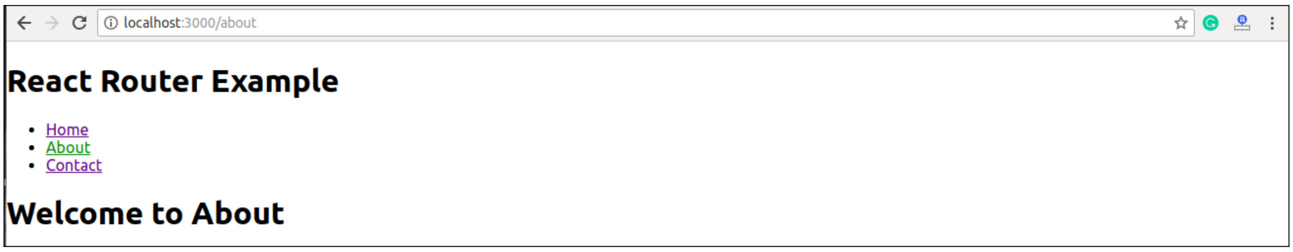
localhost:3000

React Router Example

- Home
- About
- Contact

Welcome Home

Now, when we click on **About** link, its color shown **green** that is the currently **active** link.


localhost:3000/about

React Router Example

- Home
- About
- Contact

Welcome to About

<Link> vs <NavLink>

The Link component allows navigating the different routes on the websites, whereas NavLink component is used to add styles to the active routes.

React Router Switch

The <**Switch**> component is used to render components only when the path will be **matched**. Otherwise, it returns to the **not found** component.

To understand this, first, we need to create a **notfound** component.

notfound.js

```
import React from 'react'
const Notfound = () => <h1>Not found</h1>
export default Notfound
```

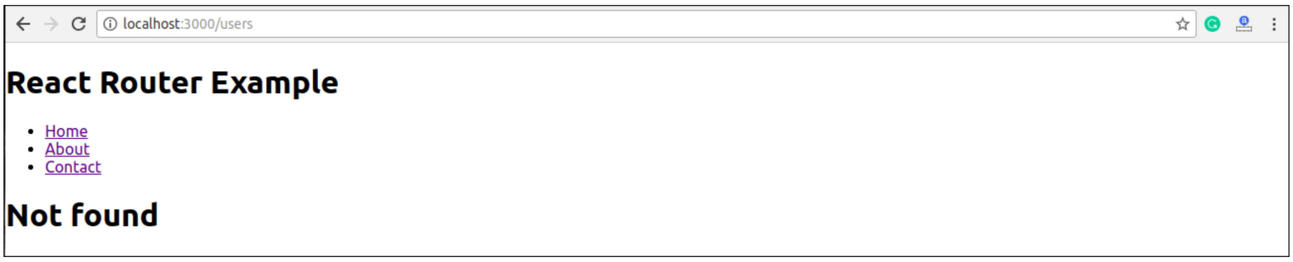
Now, import component in the index.js file. It can be seen in the below code.

Index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter as Router, Route, Link, NavLink, Switch } from 'react-router-dom'
import './index.css';
import App from './App';
import About from './about'
import Contact from './contact'
import Notfound from './notfound'

const routing = (
  <Router>
    <div>
      <h1>React Router Example</h1>
      <ul>
```

```
        <li>
          <NavLink to="/" exact activeStyle={
             {color:'red'}
          }>Home</NavLink>
        </li>
        <li>
          <NavLink to="/about" exact activeStyle={
             {color:'green'}
          }>About</NavLink>
        </li>
        <li>
          <NavLink to="/contact" exact activeStyle={
             {color:'magenta'}
          }>Contact</NavLink>
        </li>
      </ul>
      <Switch>
         <Route exact path="/" component={App} />
         <Route path="/about" component={About} />
         <Route path="/contact" component={Contact} />
         <Route component={Notfound} />
      </Switch>
    </div>
  </Router>
)
ReactDOM.render(routing, document.getElementById('root'));
```
Output

If we manually enter the **wrong** path, it will give the not found error.


localhost:3000/users

React Router Example

- Home
- About
- Contact

Not found

React Router <Redirect>

A <Redirect> component is used to redirect to another route in our application to maintain the old URLs. It can be placed anywhere in the route hierarchy.

Nested Routing in React

Nested routing allows you to render **sub-routes** in your application. It can be understood in the below example.

Example

index.js

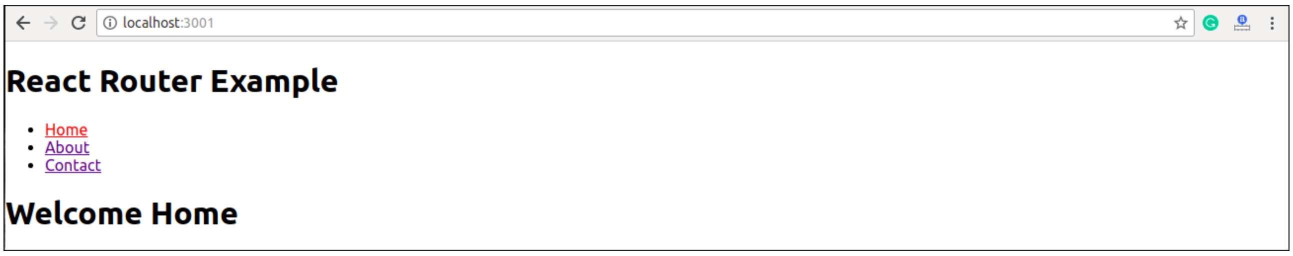
```
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter as Router, Route, Link, NavLink, Switch } from 'react-
router-dom'
import './index.css';
import App from './App';
import About from './about'
import Contact from './contact'
import Notfound from './notfound'

const routing = (
  <Router>
    <div>
      <h1>React Router Example</h1>
      <ul>
        <li>
          <NavLink to="/" exact activeStyle={
             {color:'red'}
          }>Home</NavLink>
        </li>
        <li>
          <NavLink to="/about" exact activeStyle={
             {color:'green'}
          }>About</NavLink>
        </li>
        <li>
          <NavLink to="/contact" exact activeStyle={
             {color:'magenta'}
          }>Contact</NavLink>
        </li>
      </ul>
      <Switch>
         <Route exact path="/" component={App} />
         <Route path="/about" component={About} />
         <Route path="/contact" component={Contact} />
         <Route component={Notfound} />
      </Switch>
    </div>
  </Router>
)
ReactDOM.render(routing, document.getElementById('root'));
```

contact.js

```
import React from 'react'
import { Route, Link } from 'react-router-dom'

const Contacts = ({ match }) => <p>{match.params.id}</p>

class Contact extends React.Component {
  render() {
    const { url } = this.props.match
    return (
      <div>
        <h1>Welcome to Contact Page</h1>
        <strong>Select contact Id</strong>
        <ul>
          <li>
            <Link to="/contact/1">Contacts 1 </Link>
          </li>
          <li>
            <Link to="/contact/2">Contacts 2 </Link>
          </li>
          <li>
            <Link to="/contact/3">Contacts 3 </Link>
          </li>
          <li>
            <Link to="/contact/4">Contacts 4 </Link>
          </li>
        </ul>
        <Route path="/contact/:id" component={Contacts} />
      </div>
    )
  }
}
export default Contact
```

Output

When we execute the above program, we will get the following output.

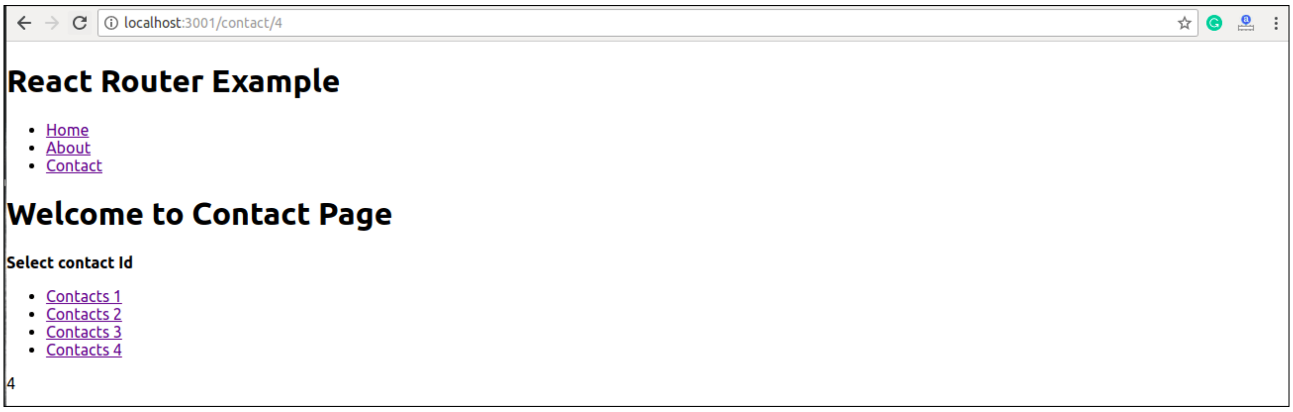

localhost:3001

React Router Example

- Home
- About
- Contact

Welcome Home

After clicking the **Contact** link, we will get the contact list. Now, selecting any contact, we will get the corresponding output. It can be shown in the below example.



localhost:3001/contact/4

React Router Example

- Home
- About
- Contact

Welcome to Contact Page

Select contact Id

- Contacts 1
- Contacts 2
- Contacts 3
- Contacts 4

4

Benefits Of React Router

The benefits of React Router is given below:

React CSS

CSS in React is used to style the React App or Component. The style attribute is the most used attribute for styling in React applications, which adds dynamically-computed styles at render time. It accepts a JavaScript object in camelCased properties rather than a CSS string. There are many ways available to add styling to your React App or Component with CSS. Here, we are going to discuss mainly four ways to style React Components, which are given below:

Inline Styling
CSS Stylesheet
CSS Module
Styled Components

1. Inline Styling

The inline styles are specified with a JavaScript object in camelCase version of the style name. Its value is the style?s value, which we usually take in a string.

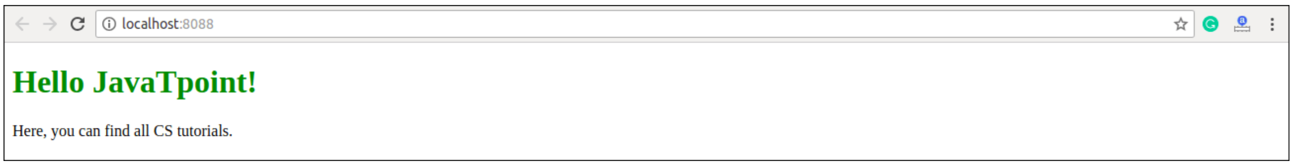
Example

App.js

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

class App extends React.Component {
  render() {
    return (
      <div>
      <h1 style={{color: "Green"}}>Hello Imarticus!</h1>
      <p>Here, you can find all CS tutorials.</p>
      </div>
    );
  }
}
export default App;
```

Note: You can see in the above example, we have used two curly braces in:
<h1 style={{color: "Green"}}>Hello Imarticus!</h1>.
It is because, in JSX, JavaScript expressions are written inside curly braces, and JavaScript objects also use curly braces, so the above styling is written inside two sets of curly braces {{}}.

Output



Hello JavaTpoint!

Here, you can find all CS tutorials.

camelCase Property Name

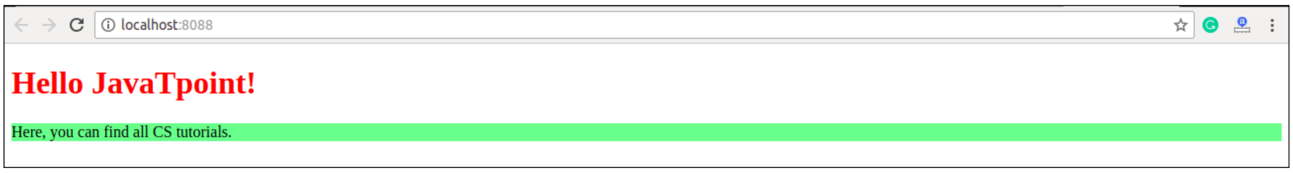
If the properties have two names, like background-color, it must be written in camel case syntax.

Example

App.js

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

class App extends React.Component {
  render() {
    return (
      <div>
      <h1 style={{color: "Red"}}>Hello Imarticus!</h1>
      <p style={{backgroundColor: "lightgreen"}}
>Here, you can find all CS tutorials.</p>
      </div>
    );
  }
}
export default App;
```

Output


localhost:8088

Hello JavaTpoint!

Here, you can find all CS tutorials.

Using JavaScript Object

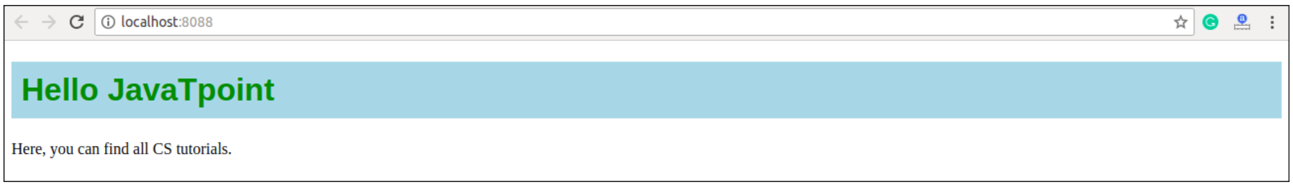
The inline styling also allows us to create an object with styling information and refer it in the style attribute.

Example

App.js

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

class App extends React.Component {
  render() {
    const mystyle = {
      color: "Green",
      backgroundColor: "lightBlue",
      padding: "10px",
      fontFamily: "Arial"
    };
    return (
      <div>
      <h1 style={mystyle}>Hello Imarticus</h1>
      <p>Here, you can find all CS tutorials.</p>
      </div>
    );
  }
}
export default App;
```

Output


localhost:8088

Hello JavaTpoint

Here, you can find all CS tutorials.

2. CSS Stylesheet

You can write styling in a separate file for your React application, and save the file with a .css extension. Now, you can import this file in your application.

Example

App.js

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

class App extends React.Component {
  render() {
    return (
      <div>
      <h1>Hello Imarticus</h1>
      <p>Here, you can find all CS tutorials.</p>
      </div>
    );
  }
}
export default App;
```

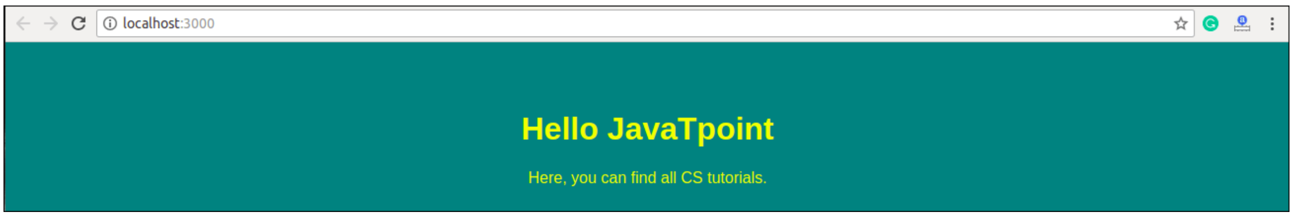
App.css

```
body {
  background-color: #008080;
  color: yellow;
  padding: 40px;
  font-family: Arial;
  text-align: center;
}
```

Index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport"
      content="width=device-width, initial-scale=1" />
    <title>React App</title>
  </head>
  <body>
    <div id="app"></div>
  </body>
</html>
```

Output


localhost:3000

Hello JavaTpoint

Here, you can find all CS tutorials.

3. CSS Module

CSS Module is another way of adding styles to your application. It is a CSS file where all class names and animation names are scoped locally by default. It is available only for the component which imports it, means any styling you add can never be applied to other components without your permission, and you never need to worry about name conflicts. You can create CSS Module with the .module.css extension like a myStyles.module.css name.

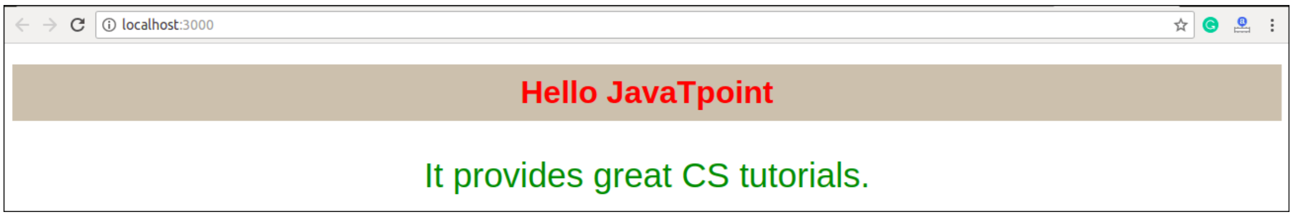
Example

App.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import styles from './myStyles.module.css';

class App extends React.Component {
  render() {
    return (
      <div>
      <h1 className={styles.mystyle}>Hello Imarticus</h1>
      <p className={styles.parastyle}>It provides great CS tutorials.</p>
      </div>
    );
  }
}
export default App;
```

myStyles.module.css

```
.mystyle {
  background-color: #cdc0b0;
  color: Red;
  padding: 10px;
  font-family: Arial;
  text-align: center;
}

.parastyle{
  color: Green;
  font-family: Arial;
  font-size: 35px;
  text-align: center;
}
```
Output


Hello JavaTpoint

It provides great CS tutorials.

4. Styled Components

Styled-components is a library for React. It uses enhance CSS for styling React component systems in your application, which is written with a mixture of JavaScript and CSS.

The styled-components provides:

Automatic critical CSS
No class name bugs
Easier deletion of CSS
Simple dynamic styling
Painless maintenance

Installation

The styled-components library takes a single command to install in your React application. which is:

```
$ npm install styled-components --save
```
Example

Here, we create a variable by selecting a particular HTML element such as <div>, <Title>, and <paragraph> where we store our style attributes. Now we can use the name of our variable as a wrapper <Div></Div> kind of React component.

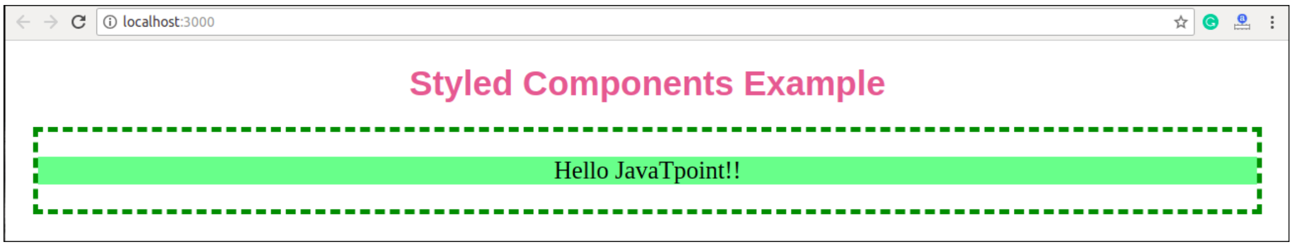
App.js

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

class App extends React.Component {
  render() {
    const Div:any = styled.div`
```

```
          margin: 20px;
          border: 5px dashed green;
          &:hover {
          background-color: ${(props:any) => props.hoverColor};
          }
          `;
    const Title = styled.h1`
          font-family: Arial;
          font-size: 35px;
          text-align: center;
          color: palevioletred;
          `;
    const Paragraph = styled.p`
          font-size: 25px;
          text-align: center;
          background-Color: lightgreen;
          `;
    return (
      <div>
          <Title>Styled Components Example</Title>
          <p></p>
          <Div hoverColor="Orange">
              <Paragraph>Hello Imarticus!!</Paragraph>
          </Div>
       </div>
    );
  }
}
export default App;
```

Output

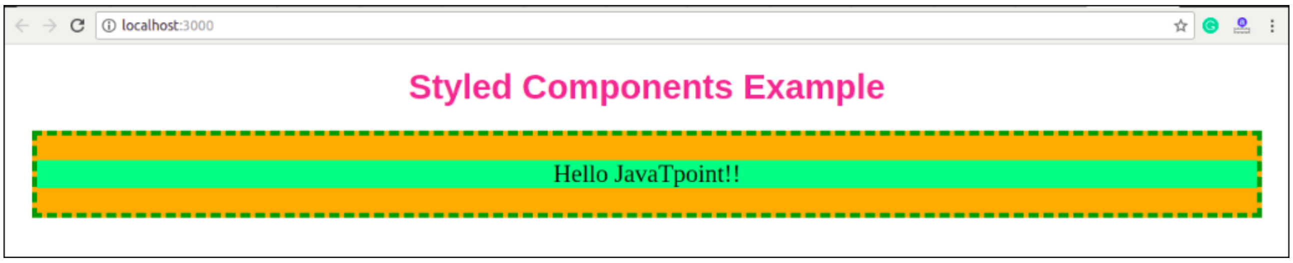
Now, execute the App.js file, we will get the output as shown below.


localhost:3000

Styled Components Example

Hello JavaTpoint!!

When we move the mouse pointer over the image, its color will be changed, as shown in the below image.



Styled Components Example

Hello JavaTpoint!!

React Bootstrap Installation

Let us create a new React app using the **create-react-app** command as follows.

```
$ npx create-react-app react-bootstrap-app
```

After creating the React app, the best way to install Bootstrap is via the npm package. To install Bootstrap, navigate to the React app folder, and run the following command.

```
$ npm install react-bootstrap bootstrap --save
```

Importing Bootstrap

Now, open the **src/index.js** file and add the following code to import the Bootstrap file.

```
import 'bootstrap/dist/css/bootstrap.min.css';
```

We can also import individual components **like import { SplitButton, Dropdown } from 'react-bootstrap';** instead of the entire library. It provides the specific components which we need to use, and can significantly reduce the amount of code.

In the React app, create a new file named **ThemeSwitcher.js** in the **src** directory and put the following code.

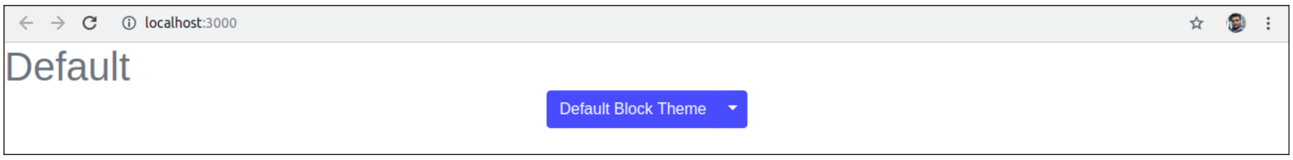
```
import React, { Component } from 'react';
import { SplitButton, Dropdown } from 'react-bootstrap';

class ThemeSwitcher extends Component {

  state = { theme: null }

  chooseTheme = (theme, evt) => {
    evt.preventDefault();
    if (theme.toLowerCase() === 'reset') { theme = null }
    this.setState({ theme });
  }

  render() {
    const { theme } = this.state;
    const themeClass = theme ? theme.toLowerCase() : 'default';

    const parentContainerStyles = {
      position: 'absolute',
      height: '100%',
      width: '100%',
      display: 'table'
    };

    const subContainerStyles = {
      position: 'relative',
      height: '100%',
      width: '100%',
      display: 'table-cell',
    };

    return (
      <div style={parentContainerStyles}>
        <div style={subContainerStyles}>

                                  <span className={`h1 center-block text-center text-$
{theme ? themeClass : 'muted'}`} style={{ marginBottom: 25 }}>{theme || 'Default'}
</span>

          <div className="center-block text-center">
                     <SplitButton bsSize="large" bsStyle={themeClass} title={`${theme ||
 'Default Block'} Theme`}>
              <Dropdown.Item eventKey="Primary Block" onSelect={this.chooseTheme
}>Primary Theme</Dropdown.Item>
              <Dropdown.Item eventKey="Danger Block" onSelect={this.chooseTheme
}>Danger Theme</Dropdown.Item>
              <Dropdown.Item eventKey="Success Block" onSelect={this.chooseThem
e}>Success Theme</Dropdown.Item>
              <Dropdown.Item divider />
               <Dropdown.Item eventKey="Reset Block" onSelect={this.chooseTheme}
>Default Theme</Dropdown.Item>
            </SplitButton>
          </div>
        </div>
      </div>
    );
  }
}
export default ThemeSwitcher;
```
Now, update the **src/index.js** file with the following snippet.

Index.js

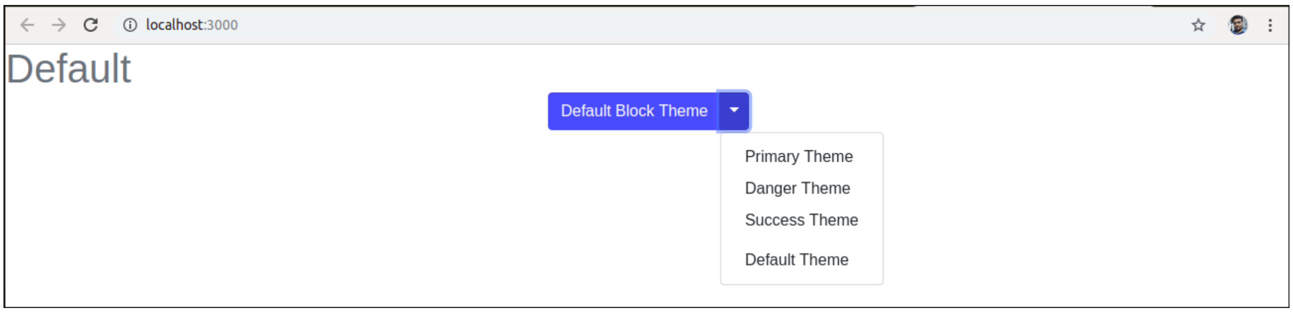
```
import 'bootstrap/dist/css/bootstrap.min.css';
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.js';
import './index.css';
import ThemeSwitcher from './ThemeSwitcher';

ReactDOM.render(<ThemeSwitcher />, document.getElementById('root'));
```

Output

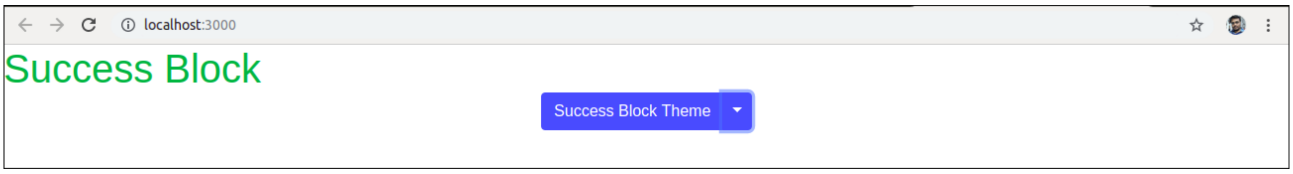
When we execute the React app, we should get the output as below.



Click on the dropdown menu. We will get the following screen.



Now, if we choose the **Success Theme**, we will get the below screen.



Let us create a new React app using the create-react-app command as follows.

```
$ npx create-react-app reactstrap-app
```

Next, install the **reactstrap** via the npm package. To install reactstrap, navigate to the React app folder, and run the following command.

```
$ npm install bootstrap reactstrap --save
```

Importing Bootstrap

Now, open the **src/index.js** file and add the following code to import the Bootstrap file.

```
import 'bootstrap/dist/css/bootstrap.min.css';
```

We can also import individual components **like import { Button, Dropdown } from 'reactstrap';** instead of the entire library. It provides the specific components which we need to use, and can significantly reduce the amount of code.

In the React app, create a new file named **ThemeSwitcher.js** in the **src** directory and put the following code.

```
import React, { Component } from 'react';
import { Button, ButtonDropdown, DropdownToggle, DropdownMenu, DropdownIt
em } from 'reactstrap';

class ThemeSwitcher extends Component {

  state = { theme: null, dropdownOpen: false }

  toggleDropdown = () => {
    this.setState({ dropdownOpen: !this.state.dropdownOpen });
  }

  resetTheme = evt => {
    evt.preventDefault();
    this.setState({ theme: null });
  }

  chooseTheme = (theme, evt) => {
    evt.preventDefault();
    this.setState({ theme });
  }
  render() {
    const { theme, dropdownOpen } = this.state;
    const themeClass = theme ? theme.toLowerCase() : 'secondary';

    return (
      <div className="d-flex flex-wrap justify-content-center align-items-center">

              <span className={`h1 mb-4 w-100 text-center text-${themeClass}`}
>{theme || 'Default'}</span>
```

```
          <ButtonDropdown isOpen={dropdownOpen} toggle={this.toggleDropdown}
>
                  <Button id="caret" color={themeClass}>{theme || 'Custom'} Theme</
Button>
         <DropdownToggle caret size="lg" color={themeClass} />
         <DropdownMenu>
                        <DropdownItem onClick={e => this.chooseTheme('Primary', e)}
>Primary Theme</DropdownItem>
                        <DropdownItem onClick={e => this.chooseTheme('Danger', e)}
>Danger Theme</DropdownItem>
                        <DropdownItem onClick={e => this.chooseTheme('Success', e)}
>Success Theme</DropdownItem>
          <DropdownItem divider />
                          <DropdownItem onClick={this.resetTheme}>Default Theme</
DropdownItem>
         </DropdownMenu>
       </ButtonDropdown>

     </div>
   );
  }
}
export default ThemeSwitcher;
```

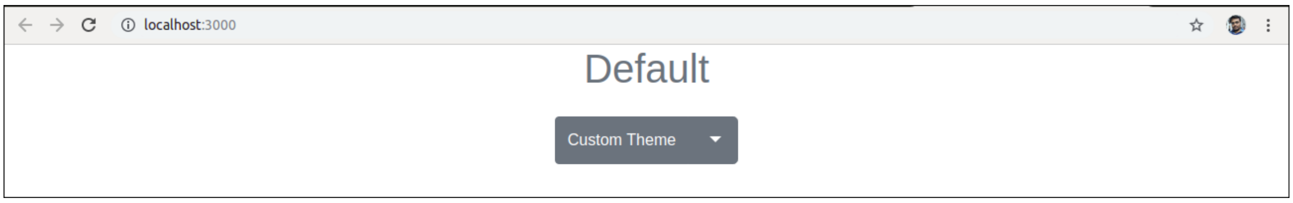
Now, update the **src/index.js** file with the following snippet.

Index.js

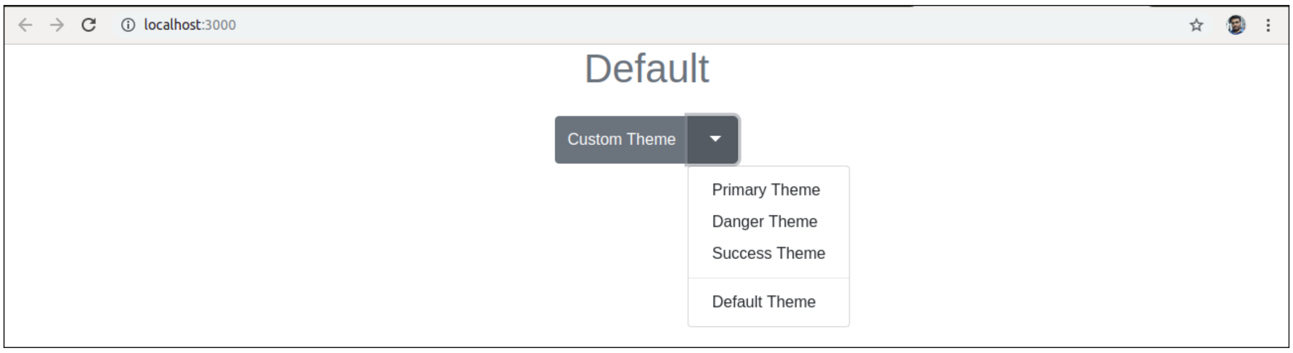
```
import 'bootstrap/dist/css/bootstrap.min.css';
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.js';
import './index.css';
import ThemeSwitcher from './ThemeSwitcher';

ReactDOM.render(<ThemeSwitcher />, document.getElementById('root'));
```

Output

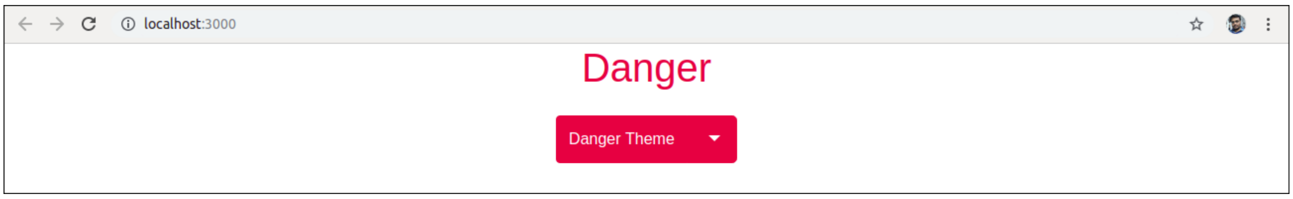
When we execute the React app, we should get the output as below.



Click on the dropdown menu. We will get the following screen.



Now, if we choose the **Danger Theme**, we will get the below screen.



React Map

A map is a data collection type where data is stored in the form of pairs. It contains a unique key. The value stored in the map must be mapped to the key. We cannot store a duplicate pair in the map(). It is because of the uniqueness of each stored key. It is mainly used for fast searching and looking up data.

```
var numbers = [1, 2, 3, 4, 5];
const doubleValue = numbers.map((number)=>{
    return (number * 2);
});
console.log(doubleValue);
```

In React, the map() method used for:

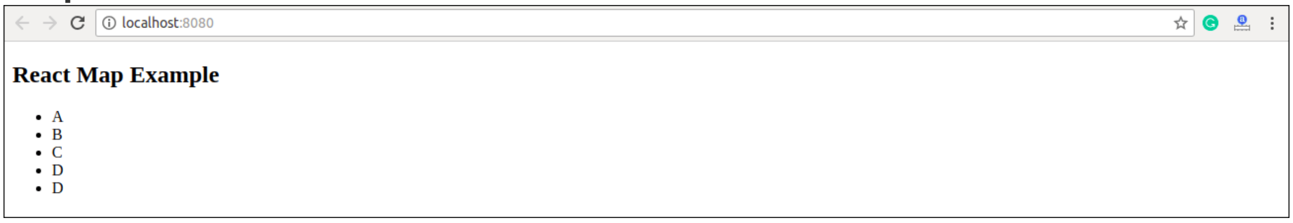
Traversing the list element.

Example

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

function NameList(props) {
  const myLists = props.myLists;
  const listItems = myLists.map((myList) =>
    <li>{myList}</li>
  );
  return (
    <div>
          <h2>React Map Example</h2>
              <ul>{listItems}</ul>
    </div>
  );
}
const myLists = ['A', 'B', 'C', 'D', 'D'];
```

```
ReactDOM.render(
  <NameList myLists={myLists} />,
  document.getElementById('app')
);
export default App;
```

Output



```
localhost:8080
```

React Map Example

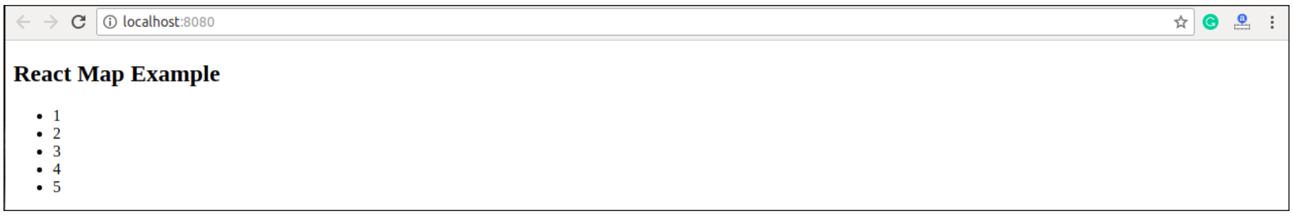
```
• A
• B
• C
• D
• D
```

2. Traversing the list element with keys.

Example

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

function ListItem(props) {
  return <li>{props.value}</li>;
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <ListItem key={number.toString()}
              value={number} />
  );
  return (
    <div>
      <h2>React Map Example</h2>
          <ul> {listItems} </ul>
    </div>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('app')
);
export default App;
```

Output


localhost:8080

React Map Example

- 1
- 2
- 3
- 4
- 5

React Table

A table is an arrangement which organizes information into rows and columns. It is used to store and display data in a structured format.

The react-table is a lightweight, fast, fully customizable (JSX, templates, state, styles, callbacks), and extendable Datagrid built for React. It is fully controllable via optional props and callbacks.

```
$ npm install react-table
```

Once, we have installed react-table, we need to **import** the react-table into the react component. To do this, open the **src/App.js** file and add the following snippet.

```
import ReactTable from "react-table";
```

Let us assume we have data which needs to be rendered using react-table.

```
const data = [{
        name: 'Ayaan',
        age: 26
        },{
        name: 'Ahana',
        age: 22
        },{
        name: 'Peter',
        age: 40
        },{
        name: 'Virat',
        age: 30
        },{
        name: 'Rohit',
        age: 32
        },{
        name: 'Dhoni',
        age: 37
        }]
```

Along with data, we also need to specify the **column info** with **column attributes**.

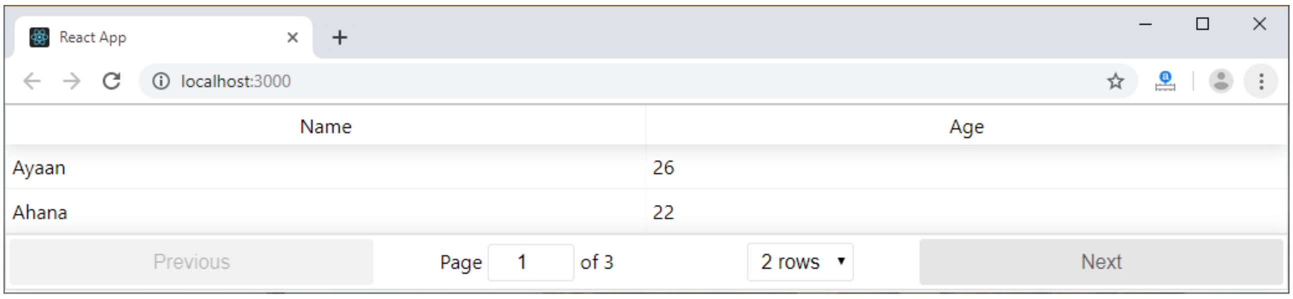
```
const columns = [{
       Header: ‘Employee Name',
       accessor: 'name'
      },{
      Header: ‘Employee Age',
      accessor: 'age'
      }]
```
Inside the render method, we need to bind this data with react-table and then returns the react-table.

```
return (
     <div>
        <ReactTable
            data={data}
            columns={columns}
            defaultPageSize = {2}
            pageSizeOptions = {[2,4, 6]}
         />
     </div>
)
```
Now, our **src/App.js** file looks like as below.

```
import React, { Component } from 'react';
import ReactTable from "react-table";
import "react-table/react-table.css";

class App extends Component {
  render() {
     const data = [{
        name: 'Ayaan',
        age: 26
        },{
         name: 'Ahana',
         age: 22
         },{
         name: 'Peter',
         age: 40
         },{
         name: 'Virat',
         age: 30
         },{
         name: 'Rohit',
         age: 32
         },{
         name: 'Dhoni',
         age: 37
         }]
     const columns = [{
       Header: 'Name',
       accessor: 'name'
       },{
       Header: 'Age',
       accessor: 'age'
       }]
   return (
       <div>
           <ReactTable
               data={data}
               columns={columns}
               defaultPageSize = {2}
               pageSizeOptions = {[2,4, 6]}
           />
       </div>
   )
  }
}
export default App;
```

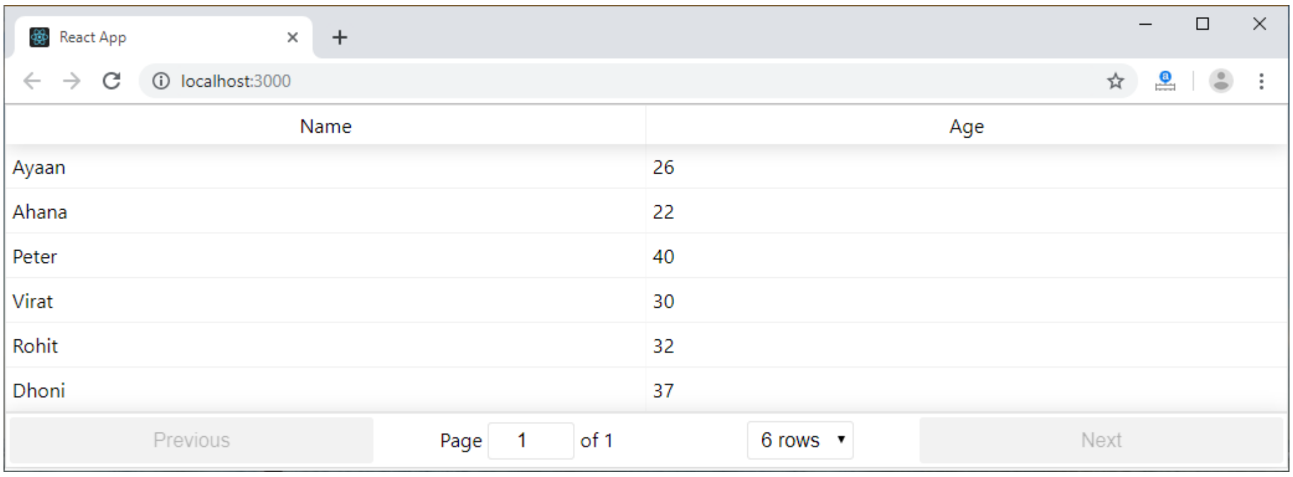
Output

When we execute the React app, we will get the output as below.



Name	Age
Ayaan	26
Ahana	22

Now, change the rows dropdown menu, we will get the output as below.



Name	Age
Ayaan	26
Ahana	22
Peter	40
Virat	30
Rohit	32
Dhoni	37