React With Redux Certification Training

COURSE OUTLINE
MODULE 02

1. Introduction to Web Development and React

2. Components and Styling the Application Layout

3. Handling Navigation with Routes

4. React State Management using Redux

5. Asynchronous Programming with Saga Middleware

6. React Hooks

7. Fetching Data using GraphQL

8. React Application Testing and Deployment

9. Introduction to React Native

10. Building React Native Applications with APIs

Topics

Following are the topics covered in this module:

- React Elements
- Render Function
- Components
- Class component
- Component constructor
- Functional components
- Multiple components
- Props
- Props with Class based and Function based component
- States
- Component lifecycle
- React Events
- React Forms
- Different Form concepts
- Styling in React
- Inline Styling
- CSS Stylesheet
- Building Music Store Application using React Components

Objectives

After completing this module, you should be able to:

- Work with React elements using Render function
- Write React applications using components
- Make use of Props to pass arguments to components
- Implement Props using States
- Manage React applications using Component Lifecycle
- Add React events in the application
- Understand different React Form concepts
- Style your React application using CSS

React Element

React Element

React element is an object *describing* DOM node and its desired properties.

It contains information about the component type (for example, a Button), its properties (for example, its colour), and any child component inside it

React elements and DOM elements are not the same. React elements are converted to DOM elements using Render function

Given below is the ***Syntax*** of writing React Elements, here we create a React element to represent ***h1 DOM*** element using ***React.createElement*** function

Type of element that we wish to create

Text to be printed or define the child component

```
React.createElement("h1", null, “Edureka");
```

element’s properties

Render Function

Render Function

Rendering is the process of transforming your **React** components into DOM nodes, that your browser can understand and display on the screen.

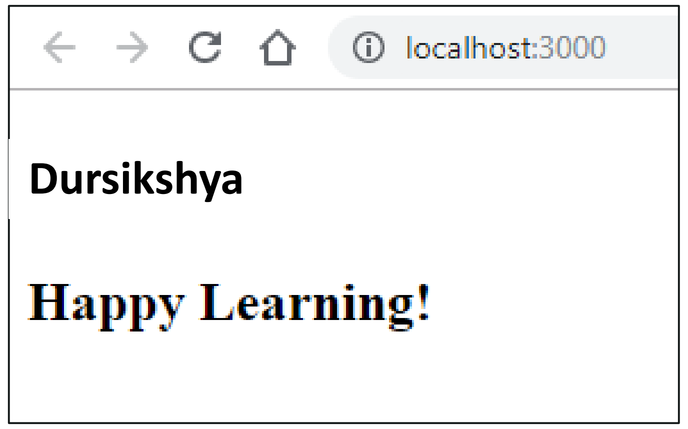
React renders a React element, including its children to the DOM via ***ReactDOM.render***

ReactDOM is the package used to access DOM in order to render React elements in the browser

Example

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

const title1 = React.createElement("h1", null, "Dursikshya");
const title2 = React.createElement("h2", null, "Happy Learning!");

ReactDOM.render([title1, title2], document.getElementById("root"));
```

Output


localhost:3000

Dursikshya

Happy Learning!

Elements to be displayed

HTML element where you want to display the result

Before React 16, it was not possible to render multiple elements.

Components

Components

Components are independent and reusable bits of code, which returns React elements that describes how a section of the UI (User Interface) should appear.

Every part of a React application is a ***component***, which ***splits*** the ***UI*** into independent reusable sections

Each independent section is ***processed separately***

We can ***easily update or change*** any component of an application without ***disturbing*** rest of the application

Components must be written in ***upper case*** to avoid ambiguity with HTML tags

Note: ***Render*** function is used only by the first component and rest make use of ***export***

Example Of Components

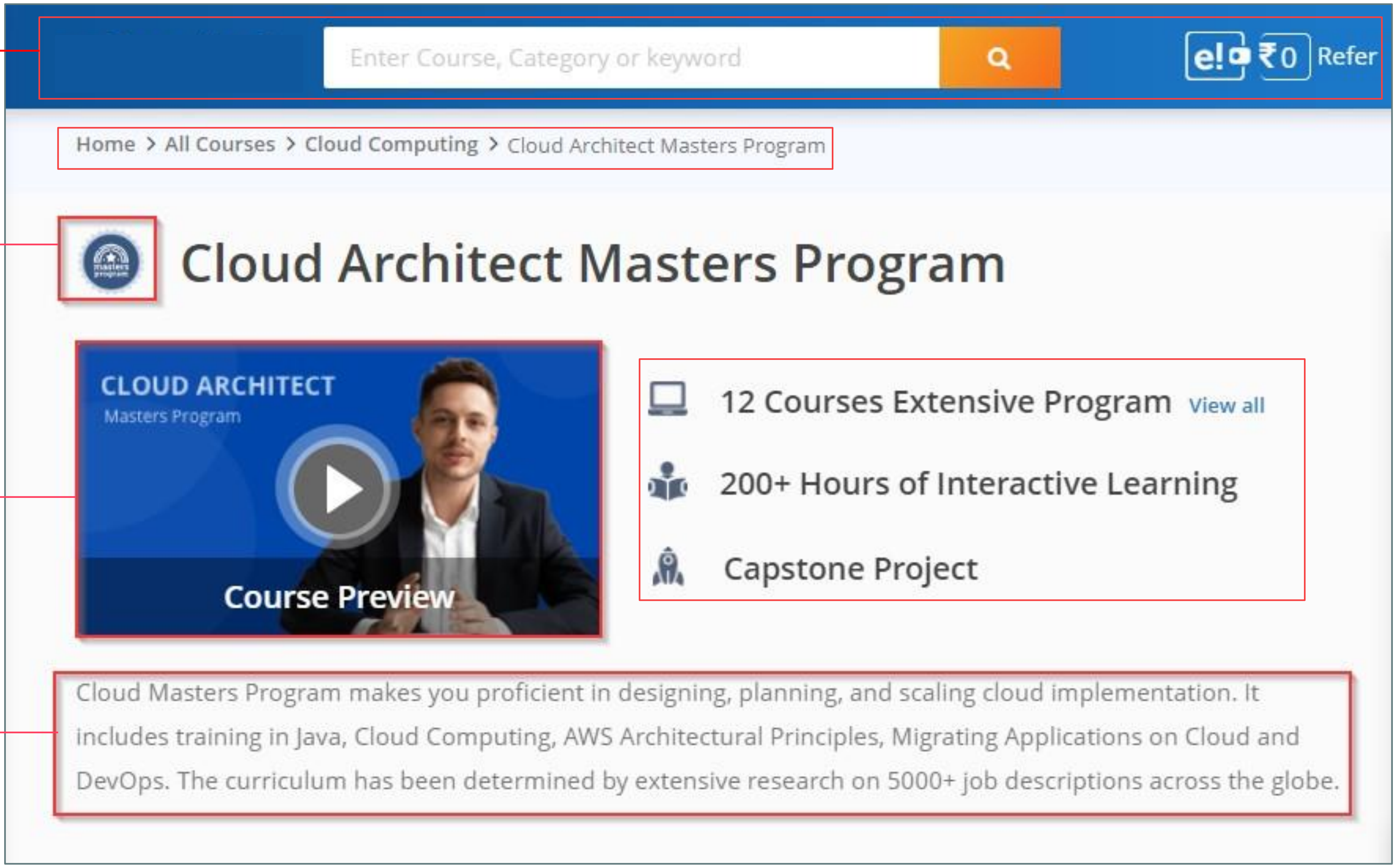
In the given below example, UI is broken down into multiple individual pieces called components, each component works independently and are merged into the parent component called as final UI.

Header Component

Image Component

Video Component

Text Component



Enter Course, Category or keyword
₹0
Refer
Home > All Courses > Cloud Computing > Cloud Architect Masters Program
Cloud Architect Masters Program
CLOUD ARCHITECT
Masters Program
Course Preview
12 Courses Extensive Program
View all
200+ Hours of Interactive Learning
Capstone Project

Cloud Masters Program makes you proficient in designing, planning, and scaling cloud implementation. It includes training in Java, Cloud Computing, AWS Architectural Principles, Migrating Applications on Cloud and DevOps. The curriculum has been determined by extensive research on 5000+ job descriptions across the globe.

There are two ways of defining a component:
Class based and Function based

Class Component

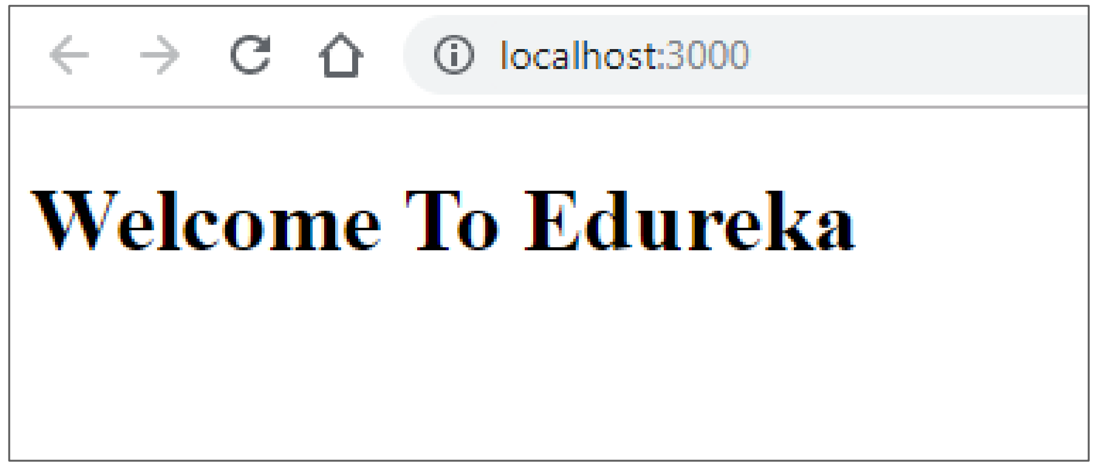
Class Component

A ***Class Component*** is defined using a *class*. It is written with '***extends React.Component'*** statement, this statement inherits ***React.Component***, and gives your component access to React.Component's functionalities.

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

class App extends React.Component {
    render() {
      return <h1>Welcome To Edureka</h1>;
    }
  }

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

- Required packages to run the application code
- Component
- Function called to provide HTML content
- HTML content to be displayed
- Component to be rendered
- HTML element

Output:



Welcome To Edureka

Use it with Component

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

class App extends Component{}
```

OR

Use it with React.Component

```
import React from 'react';

class App extends React.Component{}
```

Use Of Constructor Within The Components

01 A ***constructor*** is a member function of a class which initializes objects of a class. It has ***same name*** as the class itself

02 It is ***called automatically called*** during the creation of an object from a class

03 When implementing the constructor for a ***React.Component***, you should call ***super(props)*** before any other statement. Otherwise, ***this.props*** will be undefined to the constructor, which can lead to bugs

04 ***super()*** method is used to ***call*** the constructor of the ***parent class***

05 If your application code ***does not contain state or props*** within the component or it is not binding any ***event handlers***, then there is ***no need to define components with constructors***

Example: constructor()

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

class App extends Component {
  constructor(){
    super();
    this.state = {subject: "React with Redux Certification Training"}
  }
    render() {
      return <h2>Welcome to {this.state.subject}</h2>;
    }
  }

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

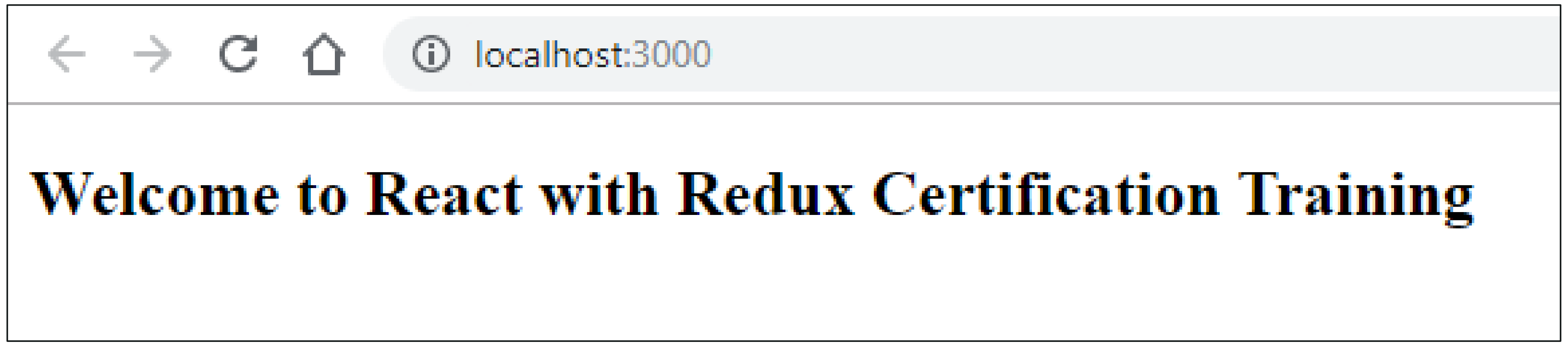
Lets App Component to receive all the functionalities provided by parent class Component

Takes props as input parameter

Informs the parent class to initiate the work

Props

Output:



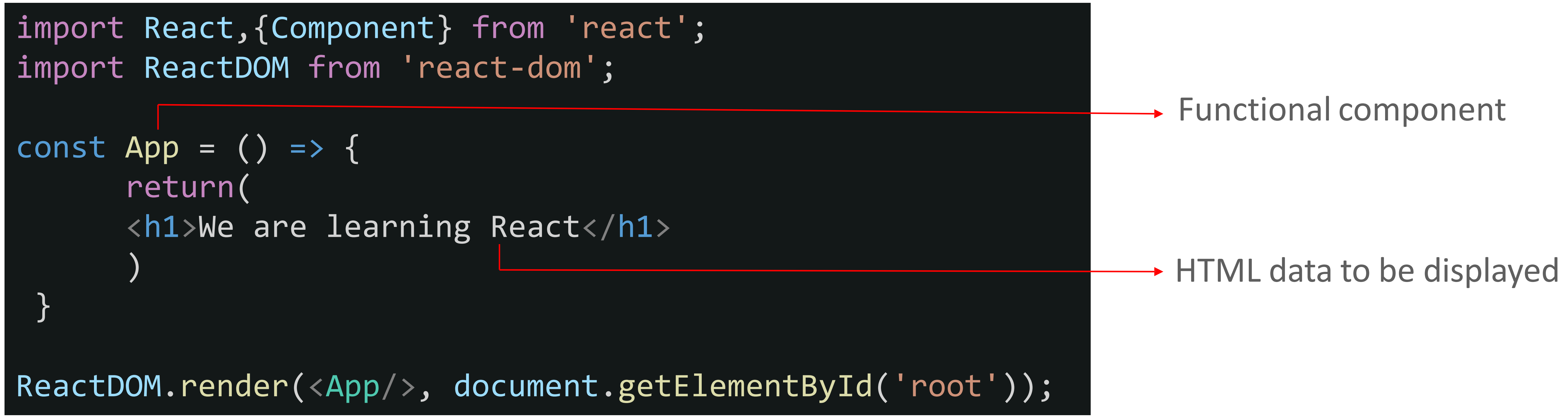
Welcome to React with Redux Certification Training

Functional Component

Functional Component

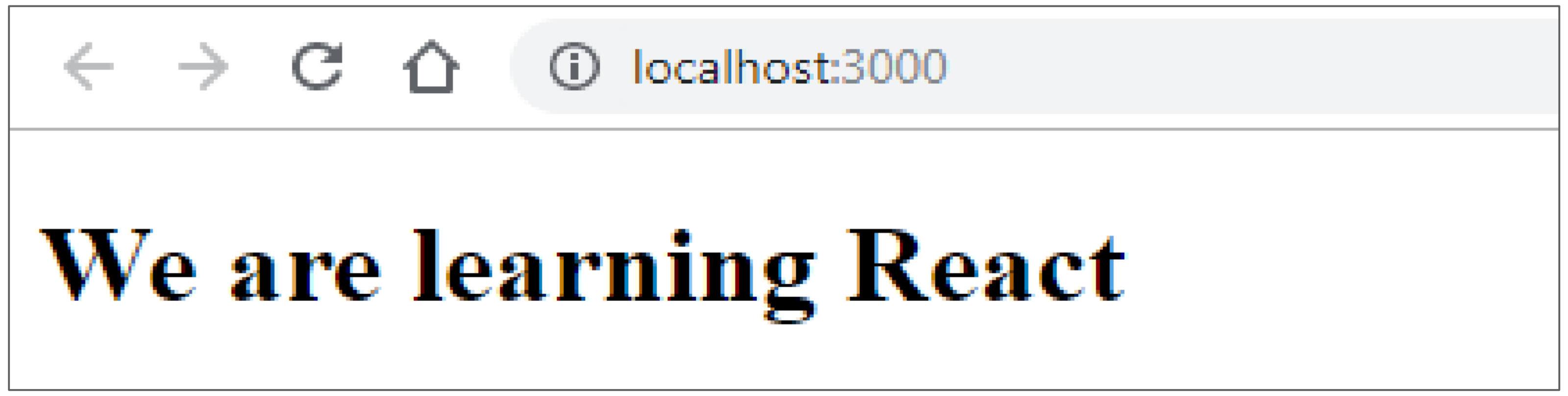
Functional components are usual JavaScript functions, which takes in ***props*** and returns ***React Element***.

Example



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

const App = () => {
      return(
      <h1>We are learning React</h1>
      )
 }

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

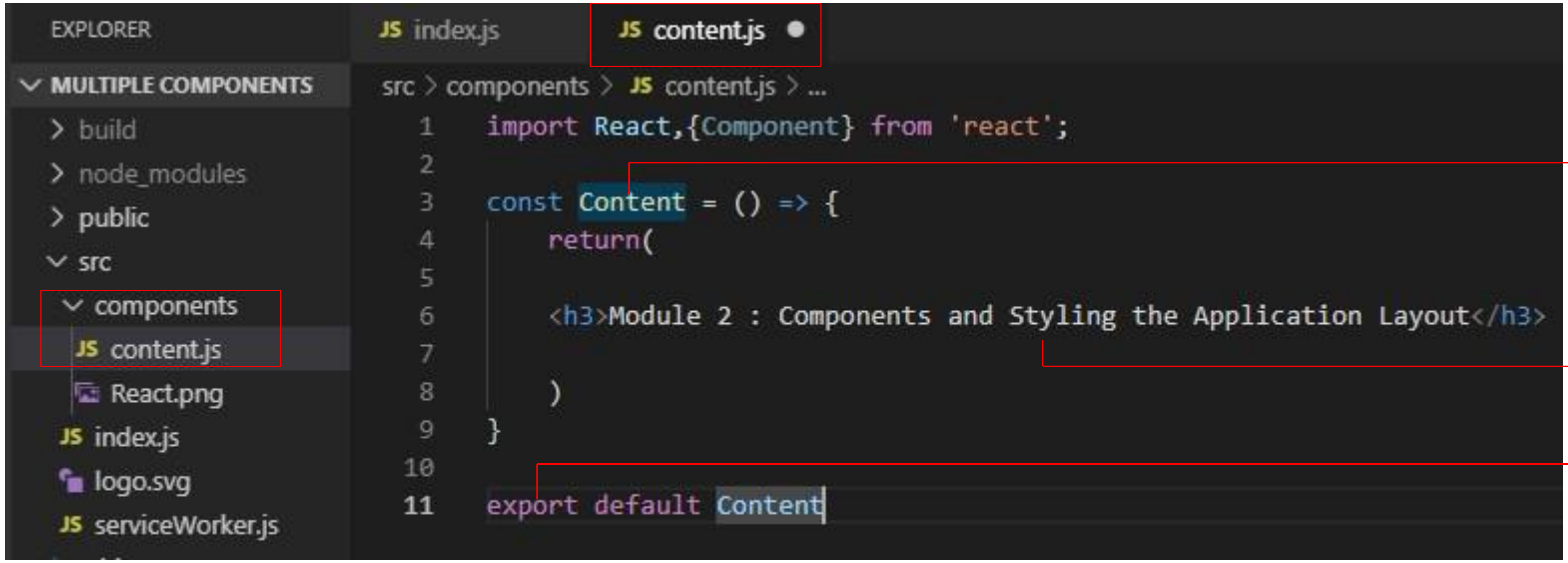
Output



We are learning React

How To Define Multiple Components

It is a good way to maintain a folder called ***components***, to add multiple components list.



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

const Content = () => {
    return(

    <h3>Module 2 : Components and Styling the Application Layout</h3>

    )
}

export default Content
```

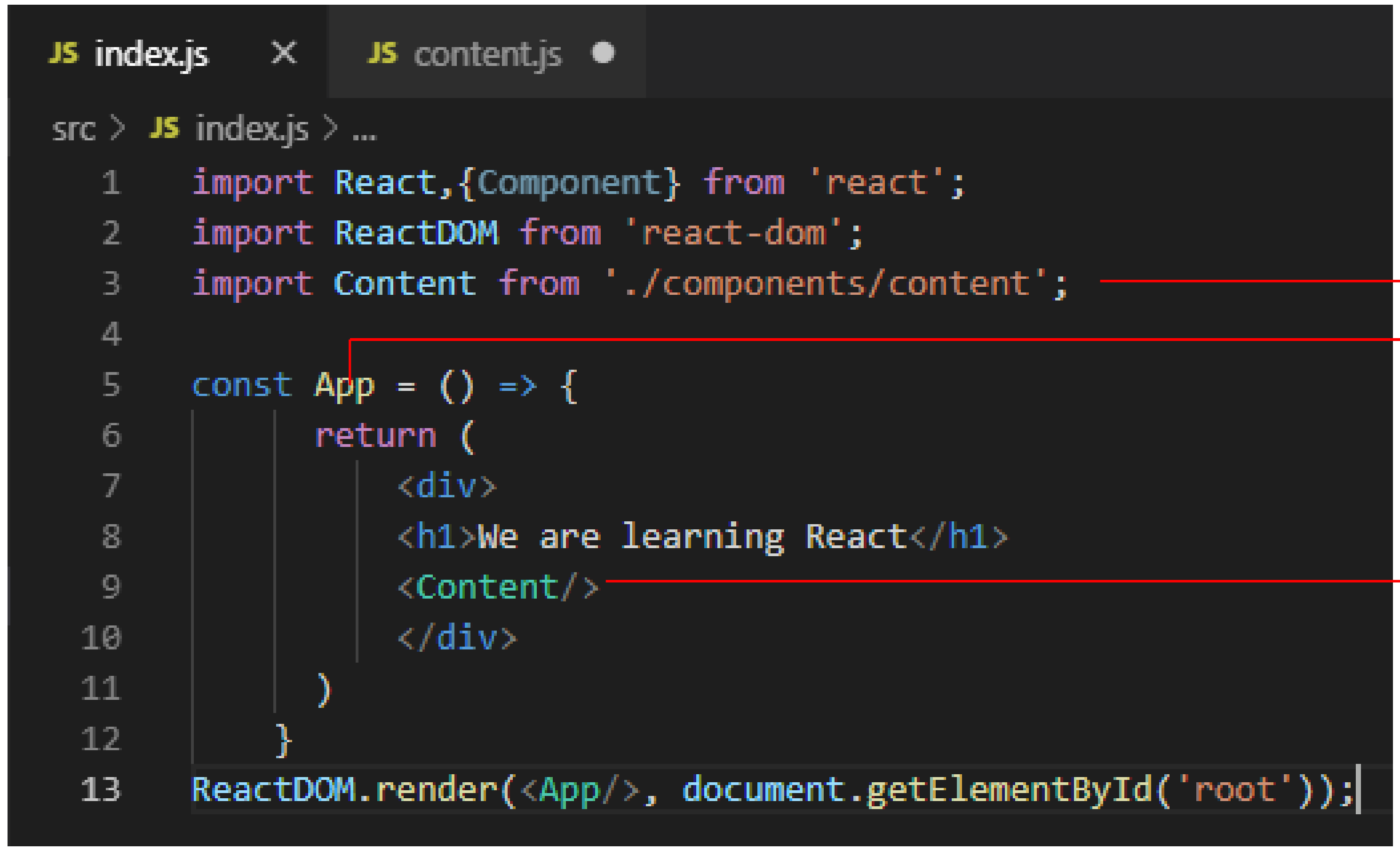
Functional component

Content to be displayed

Used to ***connect*** the Content component to the main component

How To Define Multiple Components (Contd.)

Open ***Index.js*** file and add the path of *Content* component.



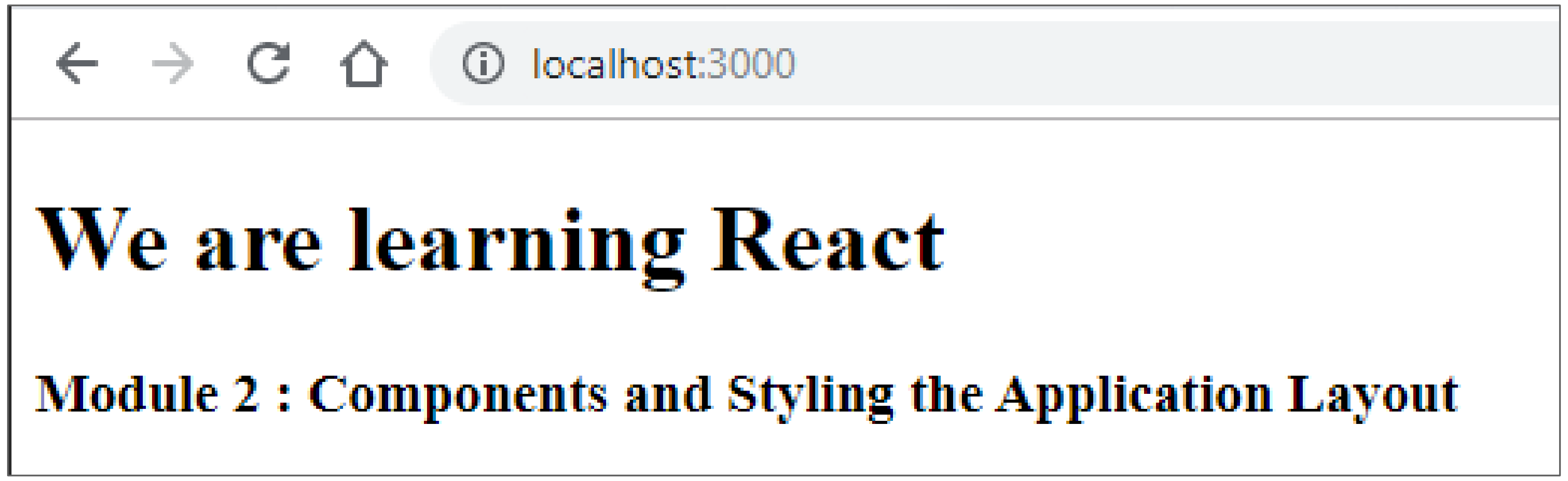
```
import React,{Component} from 'react';
import ReactDOM from 'react-dom';
import Content from './components/content';

const App = () => {
        return (
            <div>
            <h1>We are learning React</h1>
            <Content/>
            </div>
        )
    }
ReactDOM.render(<App/>, document.getElementById('root'));
```

Path of Content component (Child component)

Parent component

Displays data present in Content component (Child component)

Output:



```
We are learning React

Module 2 : Components and Styling the Application Layout
```

Functional Component Vs Class Component

Functional Component	*Class Component*
Functional components are ***simple*** to read, understand and are written in ***few lines of code***	Class components offer ***more features***, this makes the code a little bulky than Functional components
They can ***access props***, but they ***lack*** state and life cycle hence used as ***presentational components***	They are used as ***container components***, as they access props, handle state management and lifecycle
Due to lack of states, functional components are ***stateless***	Class components are ***stateful*** and make use of constructors to initialize state
They do not need ***'this'*** keyword to access props	They make use of ***'this'*** keyword to access *props*

React Components are controlled by ***Props* or *State*.**

Props

Props are the arguments passed to the React components.

Props are usually passed via ***HTML attributes*** to components, they are used by both class and functional components

They are used to render the data ***from parent component to child component***. Hence, flow of data in react is ***unidirectional***

Props are ***immutable***, that is their value cannot be changed

Syntax - passing Props:

<ReactComponent demoProp = "Hello" />

A prop named ***demoProp*** is passed to the component named ***ReactComponent*** with a value ***Hello***

Ways Of Writing Props

There are two ways to write Props:

Props with Class Based Component

- We can access props from the component's class using: ***this.props.propName***
- '***this.props***' is a global object which stores all props of a component

Props with Function Based Component

- To access a prop from a function we do not need to use the '***this***' keyword anymore
- Functional components accept props as ***parameters*** and can be ***accessed directly***

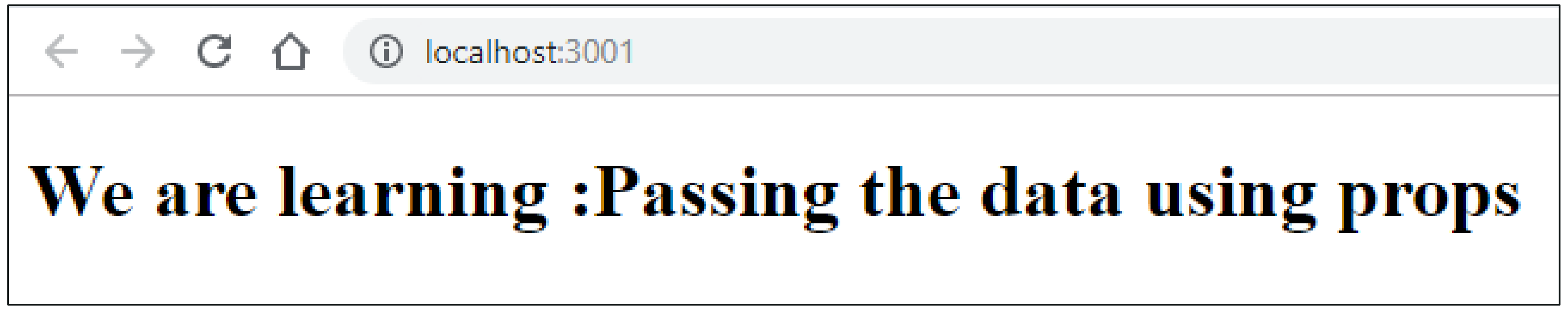
Example: Props With Class Based Component

```
import React,{Component} from 'react';
import Child from './components/child';
import ReactDOM from 'react-dom';

class Parent extends Component {
render() {

return (
 <div>
 <Child dataFromParent = "Passing the data using props"/>
 </div>

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

Class based ***Parent*** Component

Props

Output


localhost:3001

We are learning :Passing the data using props

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

class Child extends Component {
render() {

return (
     <div>
     <h1> We are learning :{this.props.dataFromParent}</h1
     </div>
);}}
export default Child
```

Class based ***Child*** Component

Accessing Props in Child Component

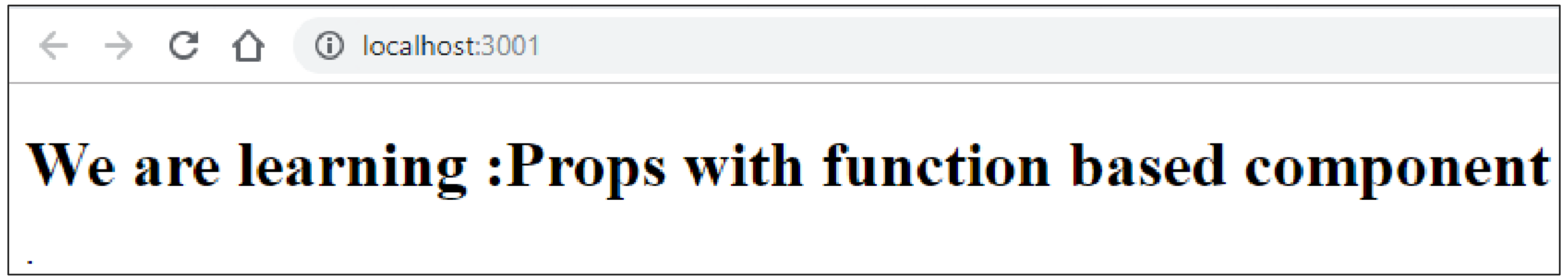
Example: Props With Function Based Component

```
import React,{Component} from 'react';
import Child from './components/child';
import ReactDOM from 'react-dom';

const Parent = () => {
return (
<div>
<Child dataFromParent = "Props with function based component"/>;
</div>
);
}
ReactDOM.render(<Parent/>, document.getElementById('root'));
```

Function based ***Parent*** Component

Props

Output


localhost:3001

We are learning :Props with function based component

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

const Child = () => {

return (
<div>
<h1> We are learning :{props.dataFromParent}</h1>
</div>
);
}
export default Child
```

Function based ***Child*** Component

Accessing Props in Child Component

Props are useful when a component is receiving data from its ***parent component,*** what should we use in-case if we have to ***receive data from some API***?
For such requirement make use of ***States***

States

States

React uses an observable object called ***state,*** to observe the changes made to the component and guide the component to behave accordingly.

01 States are ***variables*** declared within the class component which holds some information that may change over the lifetime of the component

02 They are ***mutable***, as they hold the data that change over time and controls the behaviour of the component after each change

03 They are generally updated by ***event handlers*** and are ***modified*** using ***setState()*** method

We can define state in any class as below:

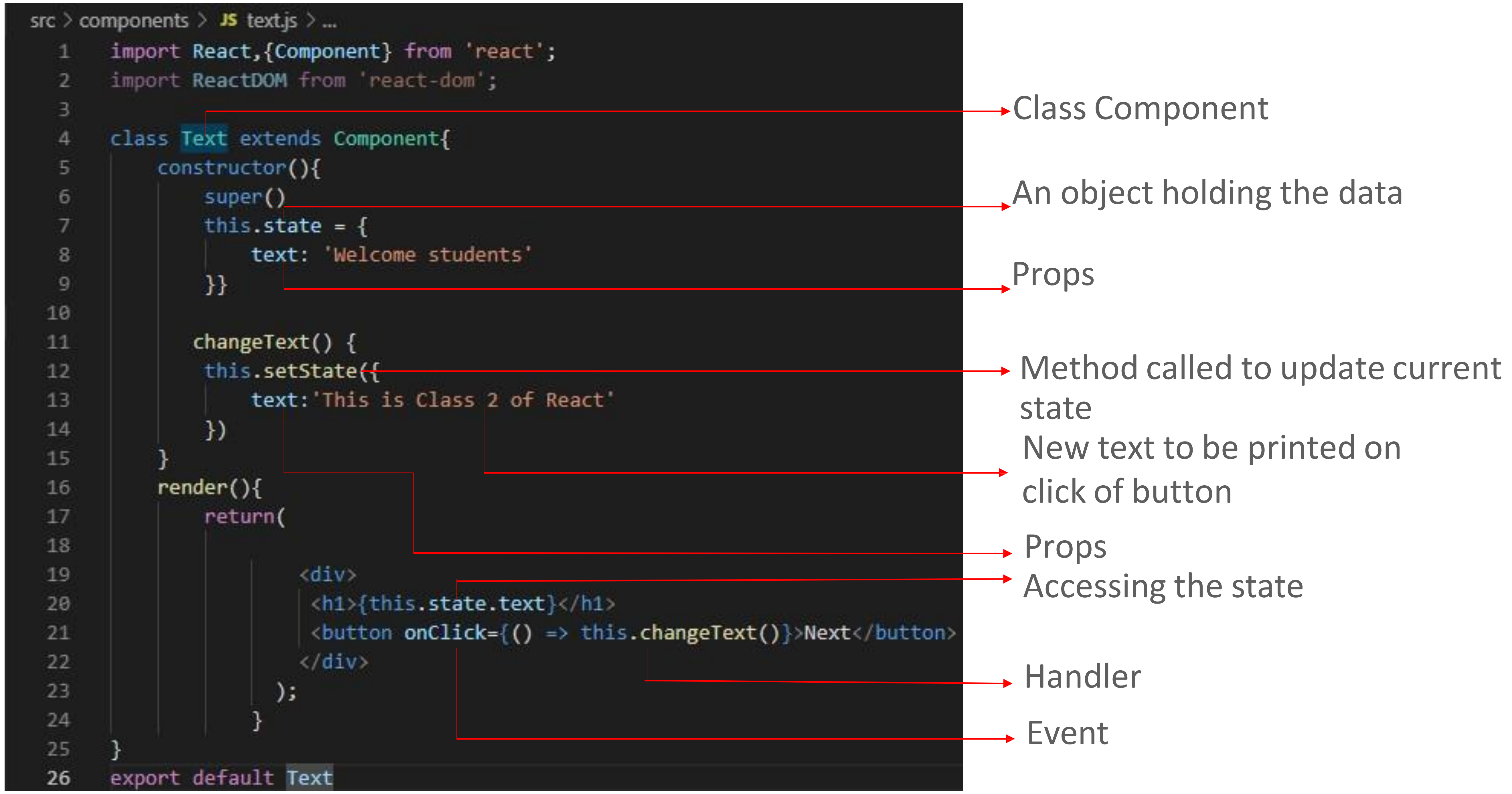
```
Class Sample extends React.Component
{
   constructor()
     {
        super();
        this.state = { attribute : "value" };
     }
}
```

Demo 1: Working Of States

Demo: Working Of States

Demo Steps

- In this demo, you will learn how to change the displayed text using state method
- Create a component called ***Text*** and add its path to the main component
- Later add the below snippet and execute the code



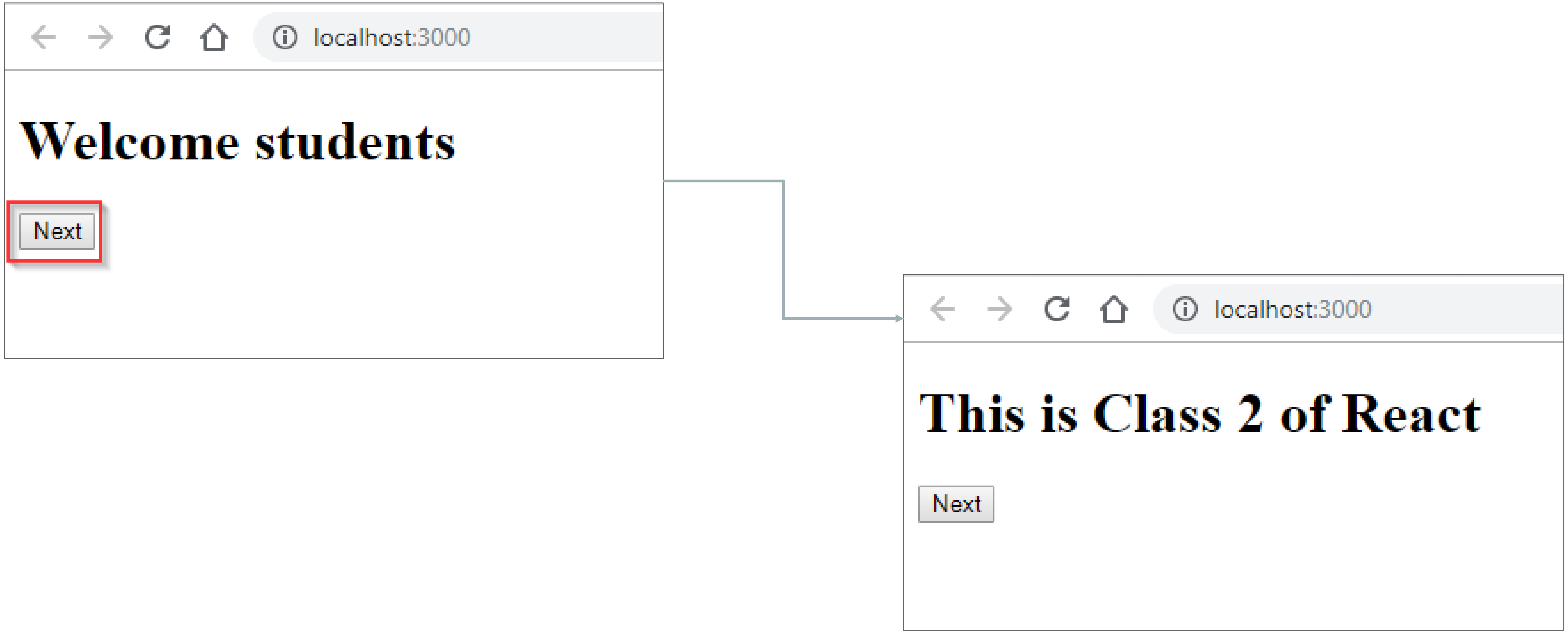
src > components > JS text.js > ...

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

class Text extends Component{
    constructor(){
        super()
        this.state = {
            text: 'Welcome students'
        }}

      changeText() {
        this.setState({
            text:'This is Class 2 of React'
        })
    }
    render(){
        return(

                <div>
                 <h1>{this.state.text}</h1>
                 <button onClick={() => this.changeText()}>Next</button>
                </div>
              );
          }
}
export default Text
```

Class Component
An object holding the data
Props
Method called to update current state
New text to be printed on click of button
Props
Accessing the state
Handler
Event

Demo: Output

Verify the working of your code with below output.



Welcome students

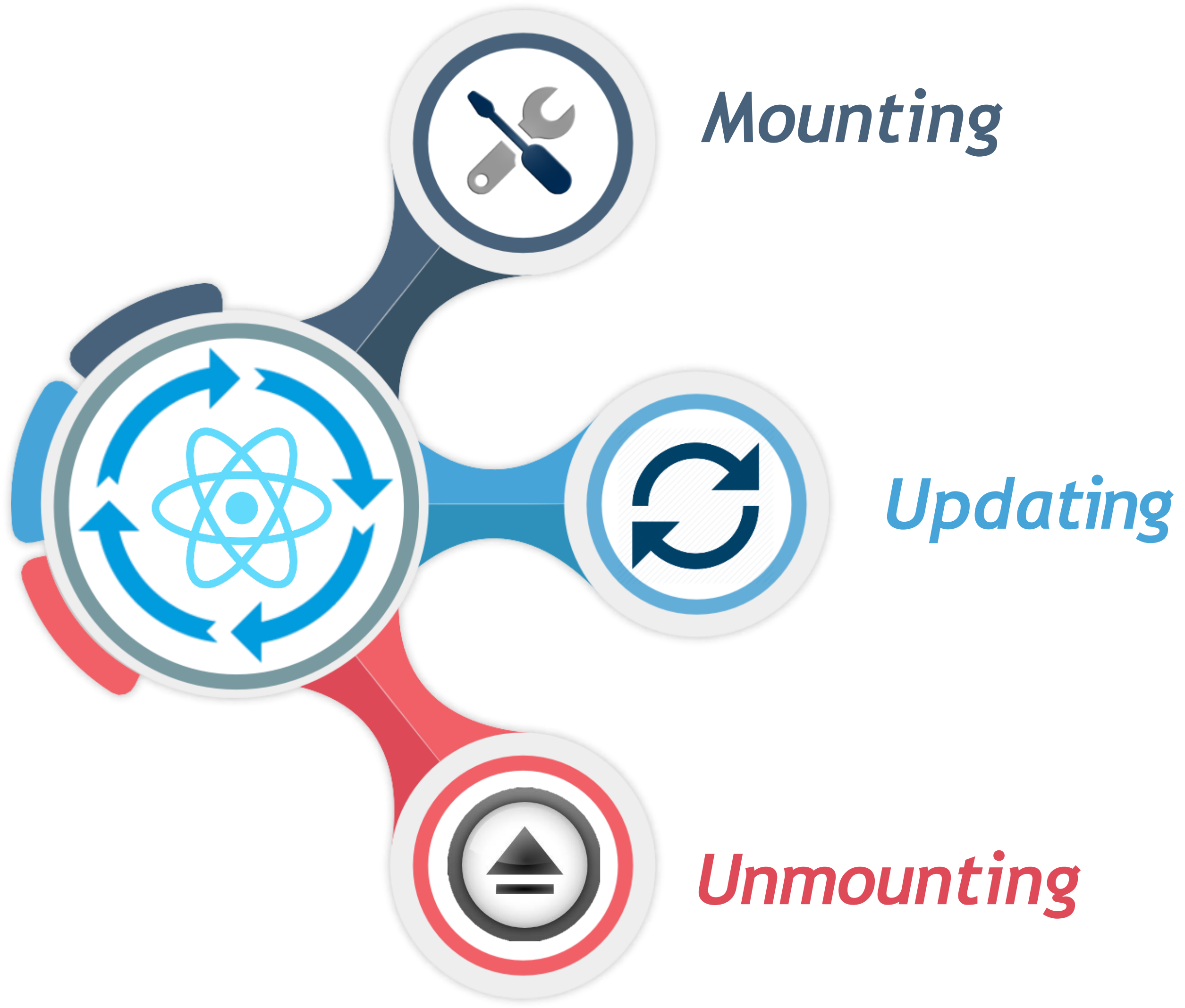
This is Class 2 of React

As we are clear with ***components, states and props,*** its time to see how they all ***work together*** to make React application user friendly.

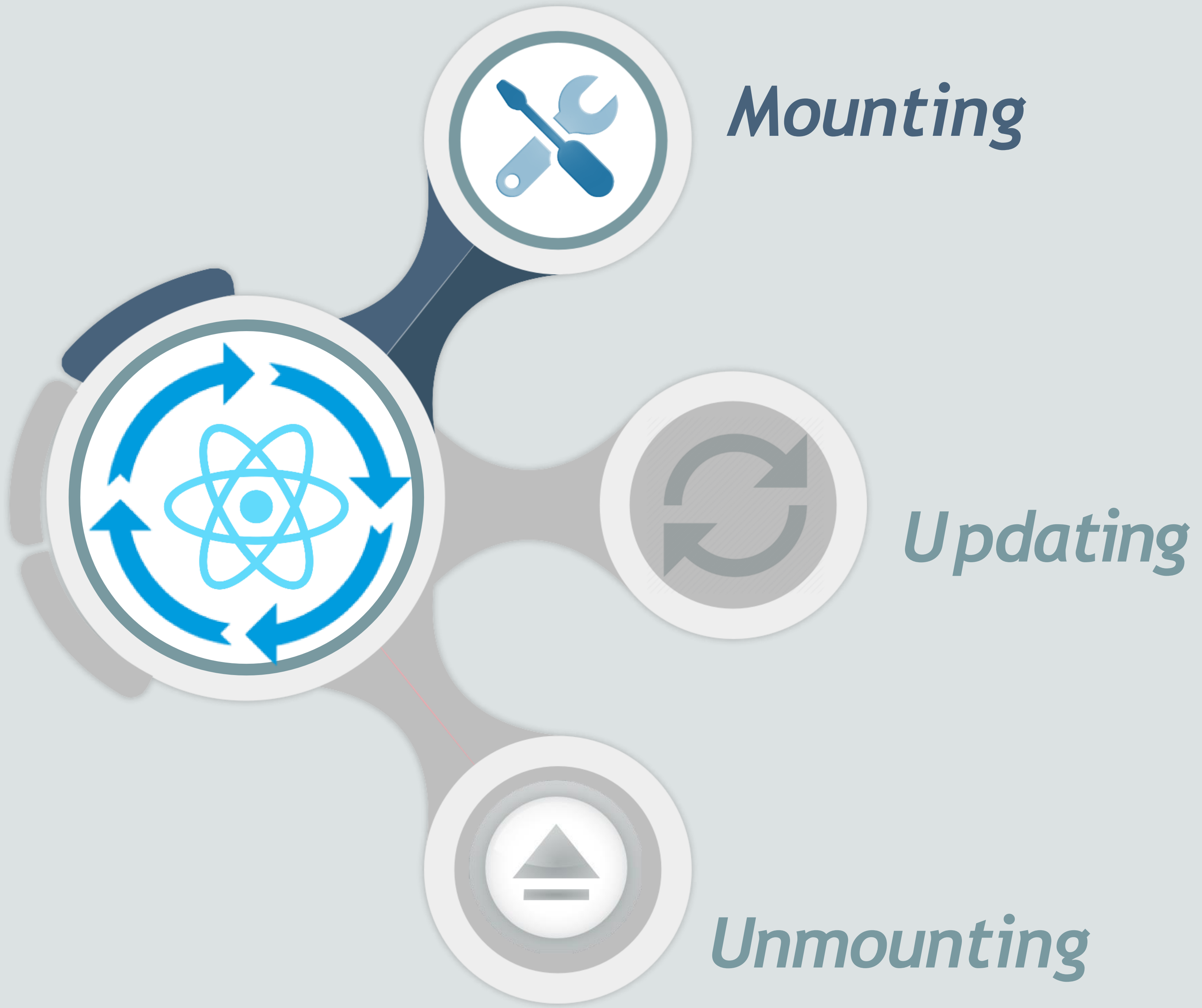
React Component Lifecycle

React Component Lifecycle

Every ***React Component follows a lifecycle***, where a series of methods are invoked in different stages.
These stages are as mentioned below:

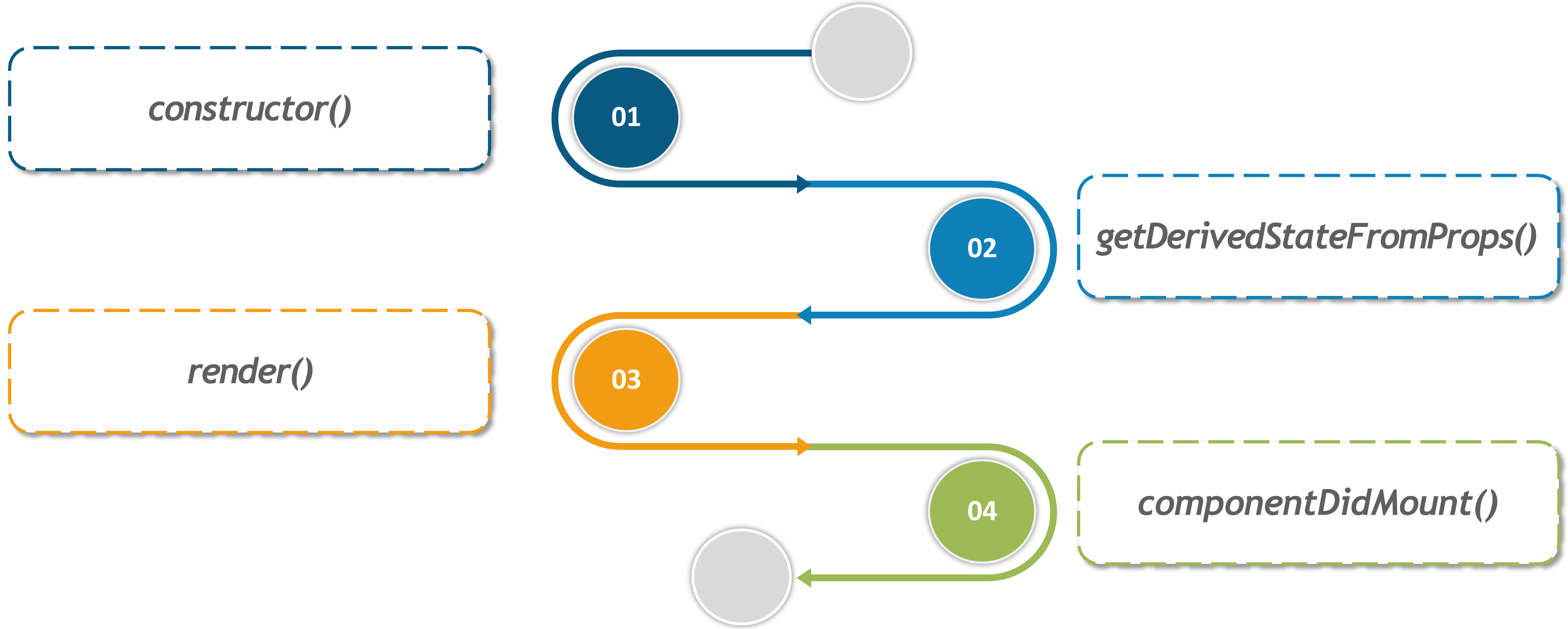


Component Lifecycle also know as Lifecycle-Hook is required when you want to control the flow of your application code.


Mounting
Updating
Unmounting

Mounting

Mounting is the phase where elements are ***added*** to DOM. During Mounting phase, four in-built methods are called simultaneously-



Mounting: constructor

When a component is initiated a ***constructor*** is called to set up the props and states within the component.

constructor()

getDerivedStateFromProps()

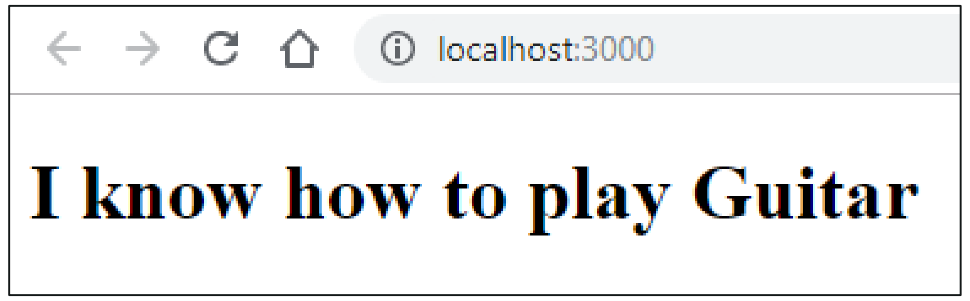
Render()

componentDidMount()

Example

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

class Music extends Component {




  render() {
   return (
   <h1>I know how to play {this.state.instrument}</h1>
   );
  }
}
ReactDOM.render(<Music />, document.getElementById('root'))
;
```

Output:



I know how to play Guitar

Mounting: getDerivedStateFromProps()

This method is called before sending the element to the DOM. It takes props and returns an object along with changes to the state. It is useful in cases where it is vital to have the previous and new value for comparison

constructor()

getDerivedStateFromProps()

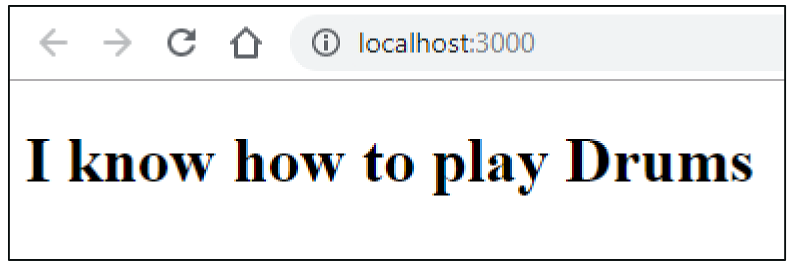
Render()

componentDidMount()

Example

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

class Music extends Component {
    constructor(props) {
      super(props);
      this.state = {instrument: "Guitar"};
    }



    render() {
      return (
        <h1>I know how to play {this.state.instrument}</h1>
      );
    }
  }
  ReactDOM.render(<Music New="Drums"/>, document.getElement
ById('root'));
```

Output:



I know how to play Drums

Above example starts with the instrument Guitar, but when the getDerivedStateFromProps() method is called, it updates the instrument based on the passed props "New"

Mounting: render()

Render() method is required to transform React Components into the DOM

constructor()

getDerivedStateFromProps()

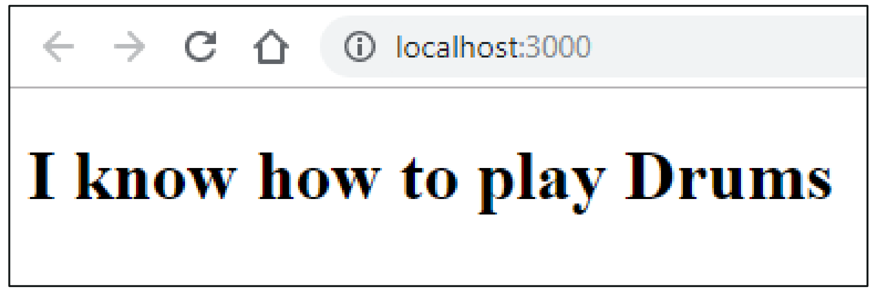
Render()

componentDidMount()

Example

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

class Music extends Component {
  constructor(props) {
    super(props);
    this.state = {instrument: "Guitar"};
  }






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

Output



I know how to play Drums

Mounting: componentDidMount()

componentDidMount() method is called when component is rendered to DOM. It confirms whether the component is placed in DOM.

constructor()

getDerivedStateFromProps()

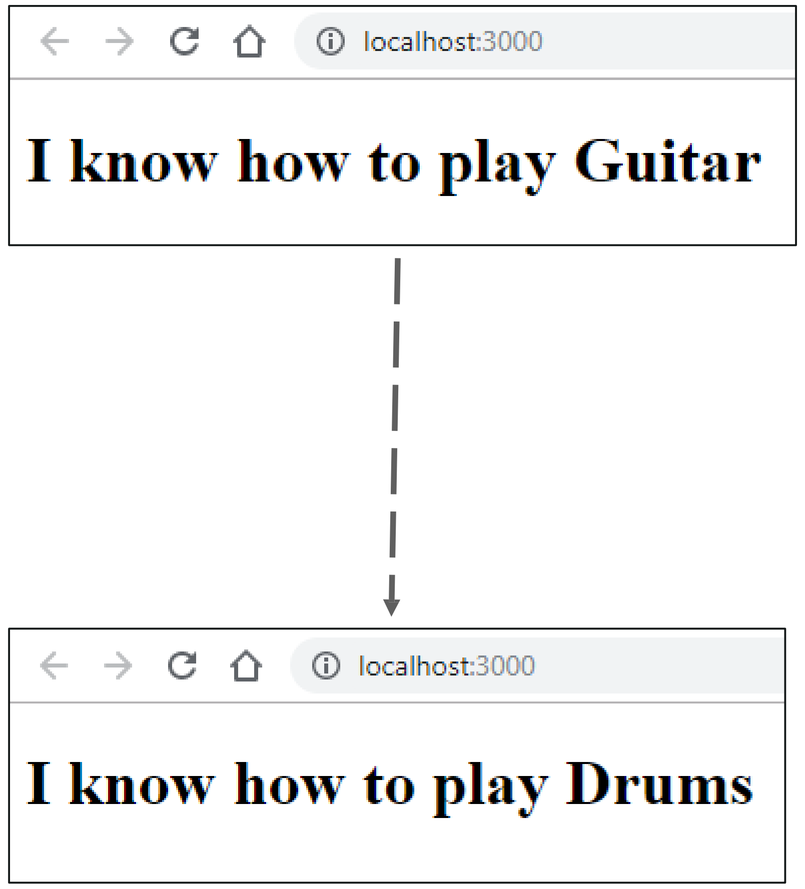
Render()

componentDidMount()

Example

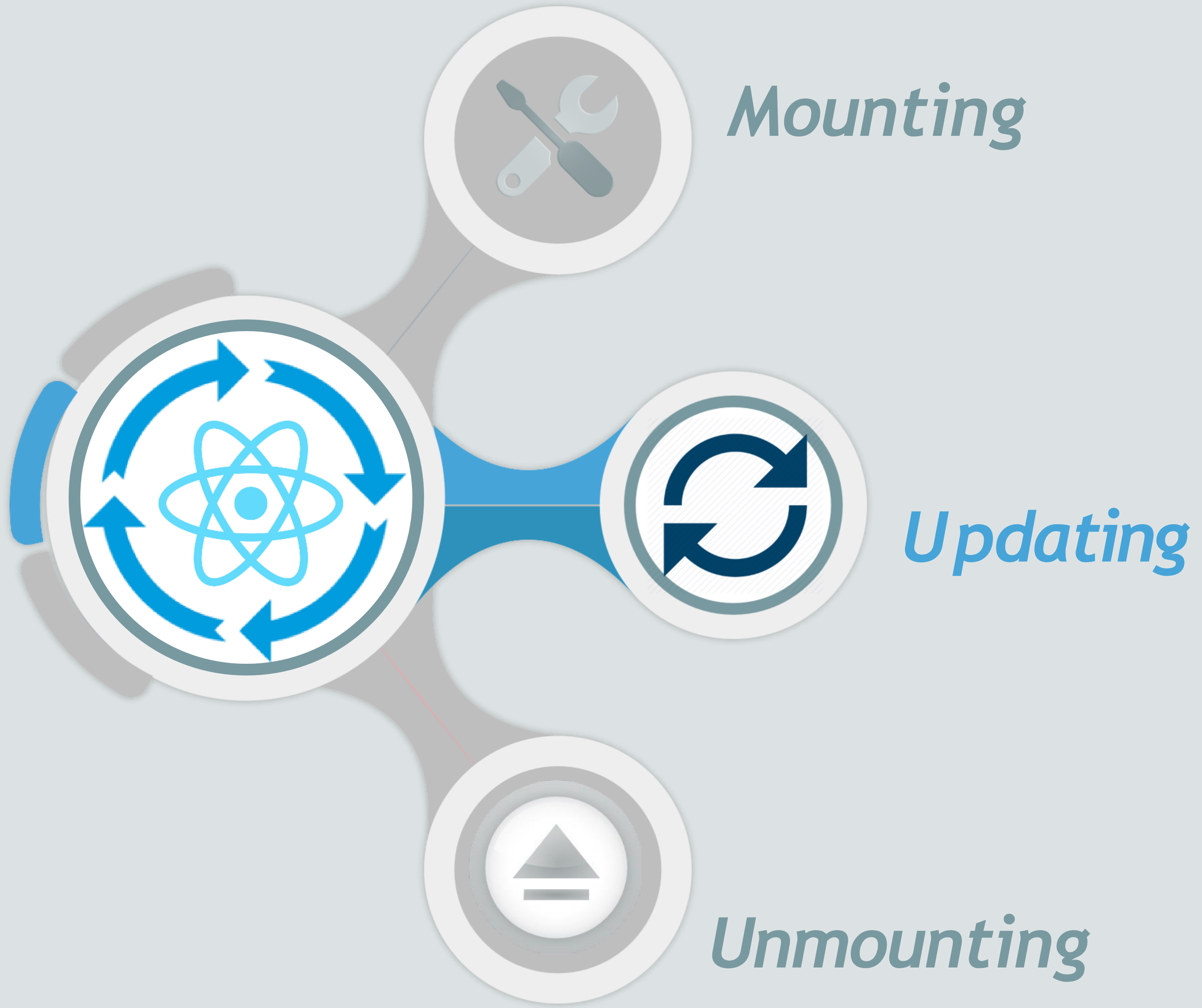
```
import React,{Component} from 'react';
import ReactDOM from 'react-dom';

class Music extends Component {
    constructor(props) {
      super(props);
      this.state = {instrument: "Guitar"};
    }





    }
    render() {
      return (
        <h1>I know how to play {this.state.instrument}</h1>
      );
    }
  }

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

Output



I know how to play Guitar

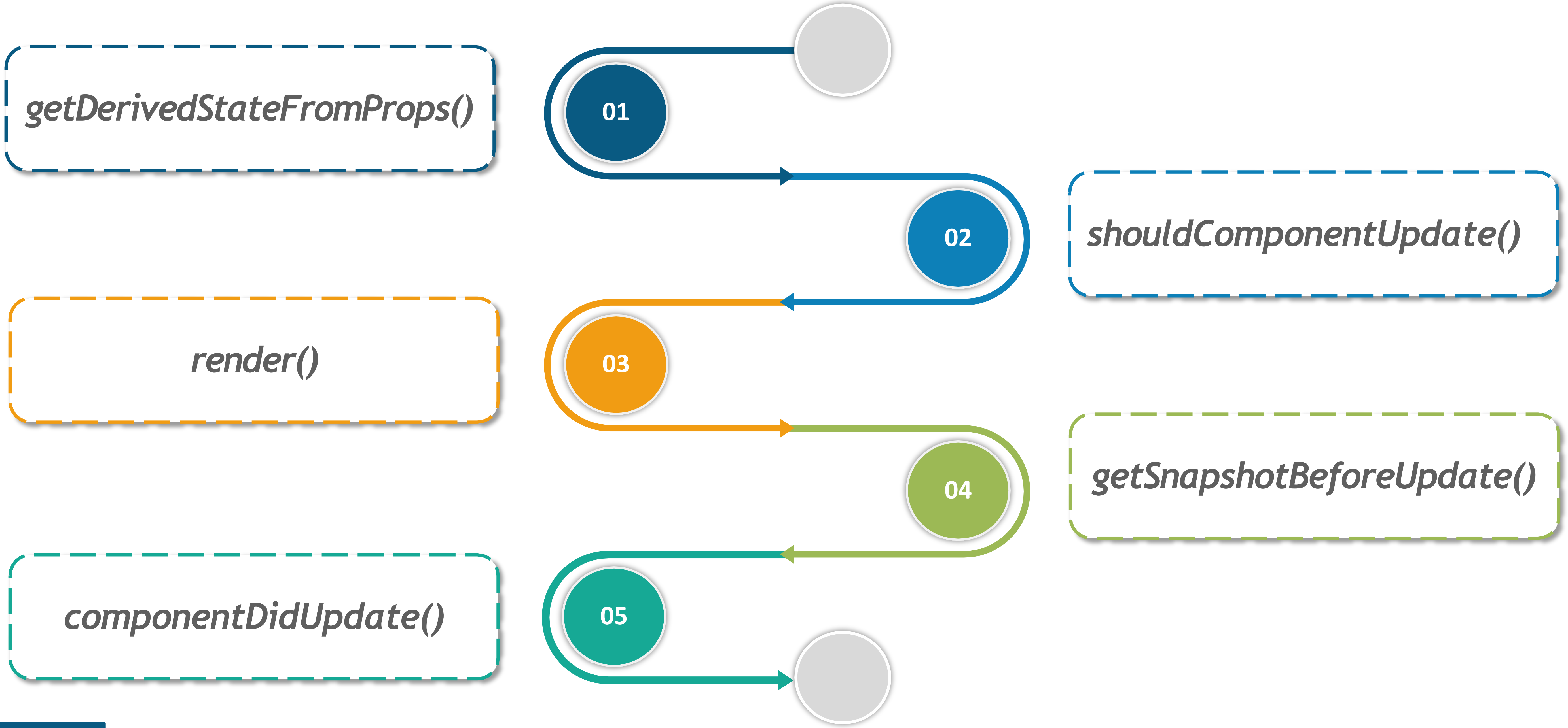
I know how to play Drums


Mounting
Updating
Unmounting

Updating

Updating is the phase where the states and props of a component are updated due to some user events such as clicking or pressing any key on keyboard. .

During Updating phase below in-built methods are called in order:



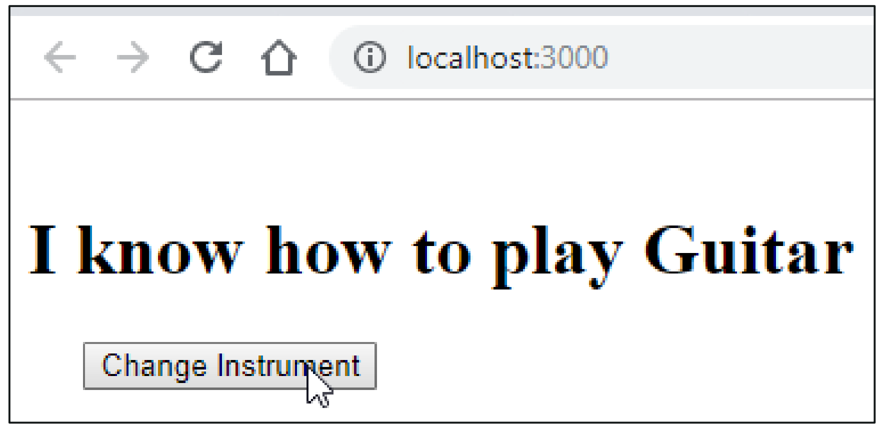
getDerivedStateFromProps() and render() methods are same as discussed before. So lets see the rest of the methods.

Updating: shouldComponentUpdate()

shouldComponentUpdate() returns a Boolean value that specifies whether React should continue with the rendering or not.

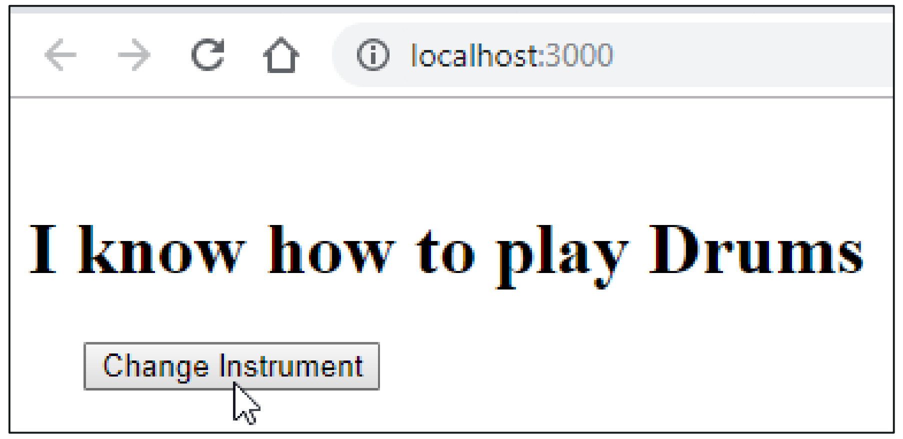
```
import React,{Component} from 'react';
import ReactDOM from 'react-dom';

class Music extends Component {
 constructor() {
  super();
  this.state = {Instrument: "Guitar"};
 }


 }
 change = () => {
  this.setState({Instrument: "Drums"});
 }
 render() {
  return (
   <div>
   <h1>I know how to play {this.state.Instrument}</h1>
   <button type="button" onClick={this.change}>Change Insrument</button>
   </div>
  );}}
ReactDOM.render(<Music />, document.getElementById('root'));
```

When function returns false, even after clicking the button, instrument does not change.


localhost:3000

I know how to play Guitar

Only when function return true, instrument changes.


localhost:3000

I know how to play Drums

Updating: getSnapshotBeforeUpdate()

getSnapshotBeforeUpdate() lets you check the values before update.
This method should include componentDidUpdate() to avoid error notifications

Example (Refer next slide for example):

01 Initially when component was ***mounting*** it rendered ***Guitar***

02 Later when component is ***mounted***, after ***completion of timer*** the instrument value changed to ***Drums***

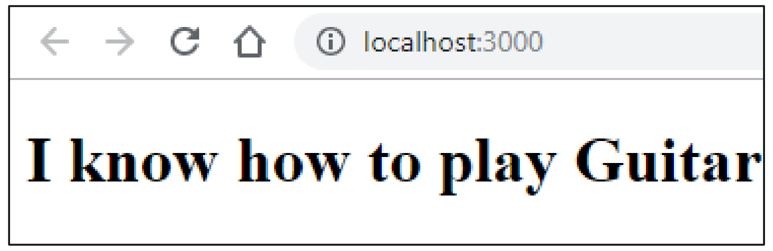
03 This ***action triggers*** the ***update phase*** and ***getSnapshotBeforeUpdate*** method is called, which writes a previous state message to the container ***CON1***

04 Then the ***componentDidUpdate()*** method is executed which writes a current state message in the container ***CON2***

Updating: getSnapshotBeforeUpdate() (Example)

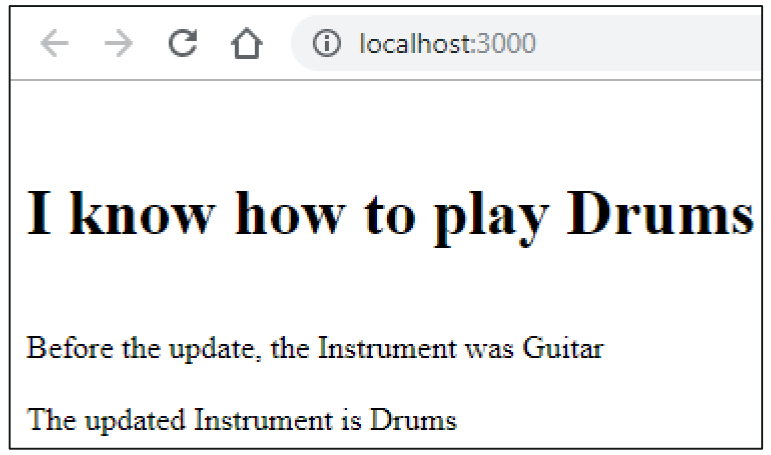
```
import React,{Component} from 'react';
import ReactDOM from 'react-dom';

class Music extends Component {
    constructor(props) {
      super(props);
      this.state = {Instrument : "Guitar"};
    }
    componentDidMount() {
      setTimeout(() => {this.setState({Instrument: "Drums"})}, 2000)
    }



    componentDidUpdate() {
      document.getElementById("CON2").innerHTML = "The updated Instrument is " + this.state.Instrument;
    }
    render() {
      return (
        <div>
          <h1>I know how to play {this.state.Instrument}</h1>
          <div id="CON1"></div>
          <div id="CON2"></div>
        </div>
      ); }}
  ReactDOM.render(<Music />, document.getElementById('root'));
```

Output: While mounting the component



I know how to play Guitar

Output: After completion of timer



I know how to play Drums

Before the update, the Instrument was Guitar

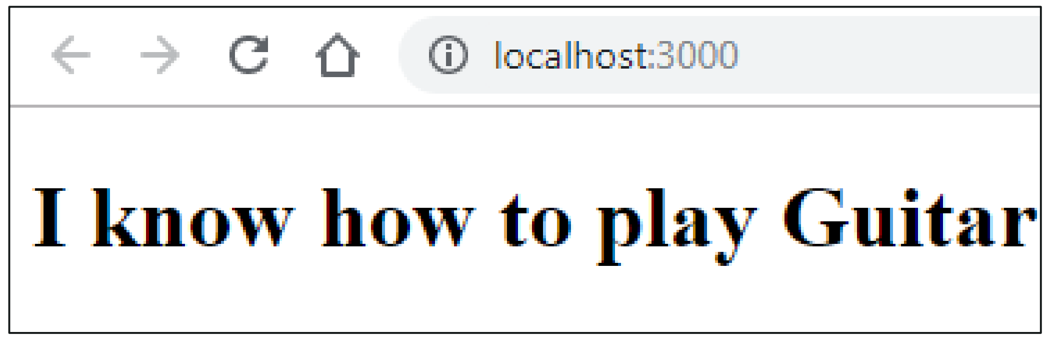
The updated Instrument is Drums

Updating: componentDidUpdate()

The ***componentDidUpdate()*** method is called after the component is updated in the DOM to verify the changes done to the DOM.

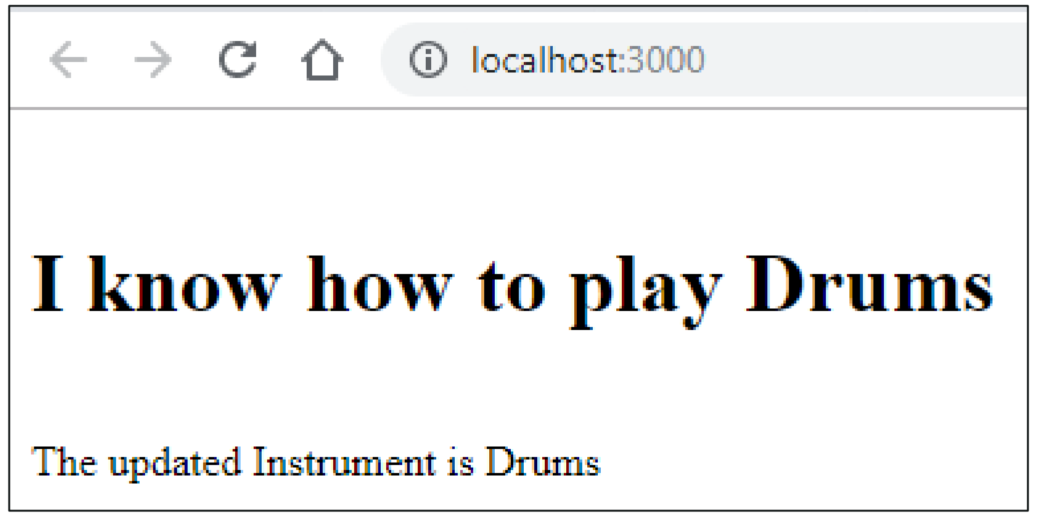
```
class Music extends Component {
  constructor(props) {
  super(props);
   this.state = {Instrument: "Guitar"};
  }
  componentDidMount() {
   setTimeout(() => {
     this.setState({Instrument: "Drums"})}, 2000)
  }



  render() {
   return (
     <div>
     <h1>I know how to play {this.state.Instrument}</h1>
     <div id="CON"></div>
     </div>
   );}}

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

Output: While mounting the component

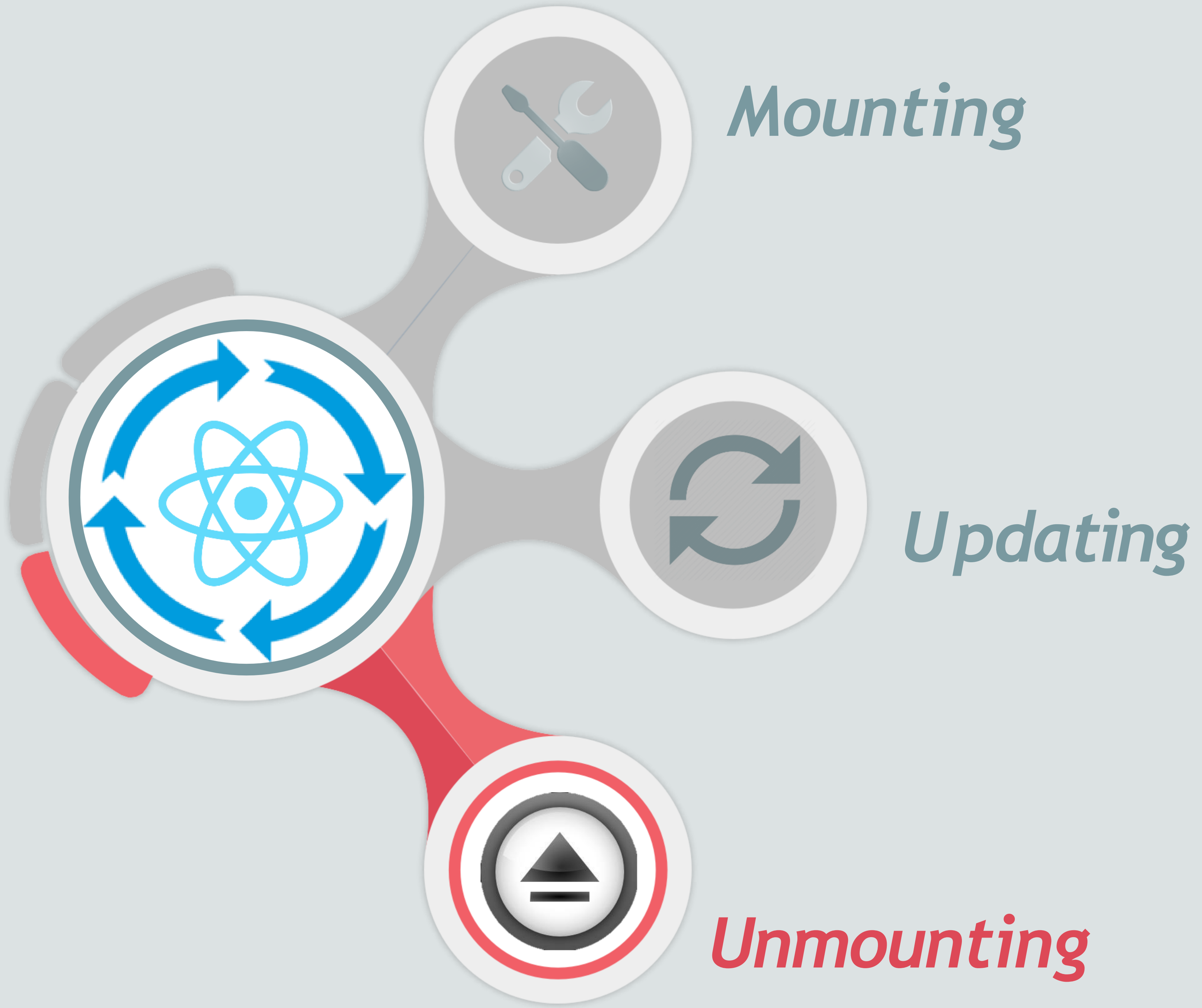


```
localhost:3000
I know how to play Guitar
```

Output: After execution of componentDidUpdate() state



```
localhost:3000
I know how to play Drums

The updated Instrument is Drums
```


Mounting
Updating
Unmounting

Unmounting

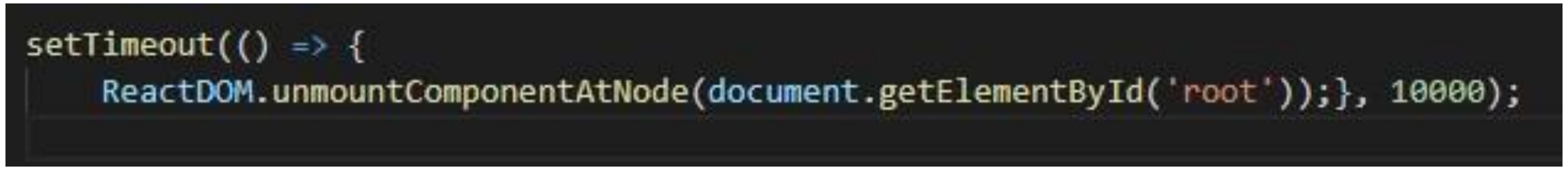
Unmounting is the phase where component is supposed to be removed from the DOM.

01 ***componentWillUnmount()*** is the only method used to remove component from the DOM

02 ***ReactDOM.render(<Component />, container)*** is the usual method of adding components to DOM, to unmount that Component from the container clean up all the attached event handlers and state

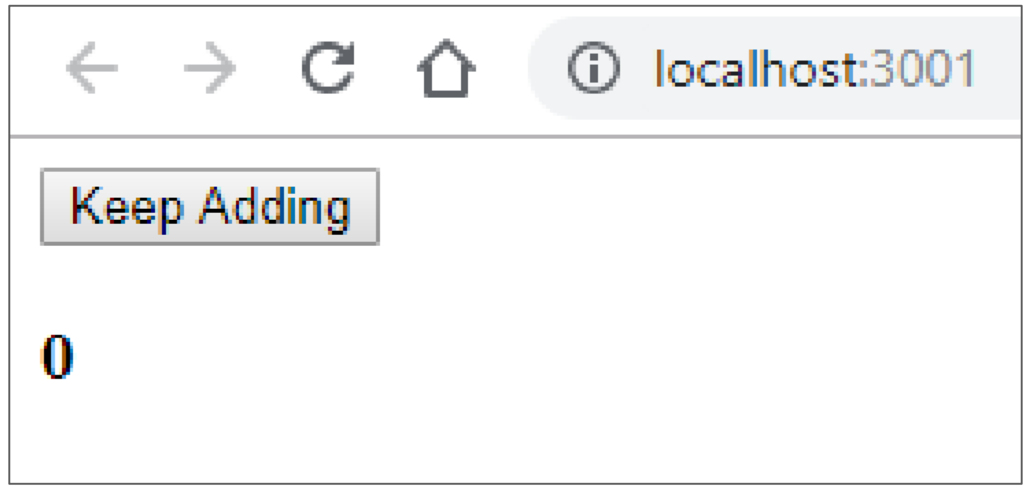
03 For unmounting the component you can make a call to {***React.unmountComponentAtNode(container_name)}***

Example:



```
setTimeout(() => {
    ReactDOM.unmountComponentAtNode(document.getElementById('root'));}, 10000);
```

Here every time on clicking the button the component is incremented by 1, after 10000 sec the component is unmounted



Demo 2: Unmounting

Refer M2 Demo Document 2 on LMS for more detailed steps

React Events

React Events

Similar to HTML, React ***executes actions*** based on ***user events***, these events mainly include: ***click, change, mouseover*** and many more

React events are written in camelCase and event handlers are written inside curly braces-
Example: ***<button onClick={this.click}>click here</button>***

Using ***JSX*** you pass a function as the event handler in place of a string

It is a good practice to always put the event handler as a ***method*** in the ***component class***

React Events: Example

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

class Event extends Component {
  click() {
   alert("Good One");
  }
  render() {
   return (
    <button onClick={this.click}>click here</button>
   );
  }
}

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

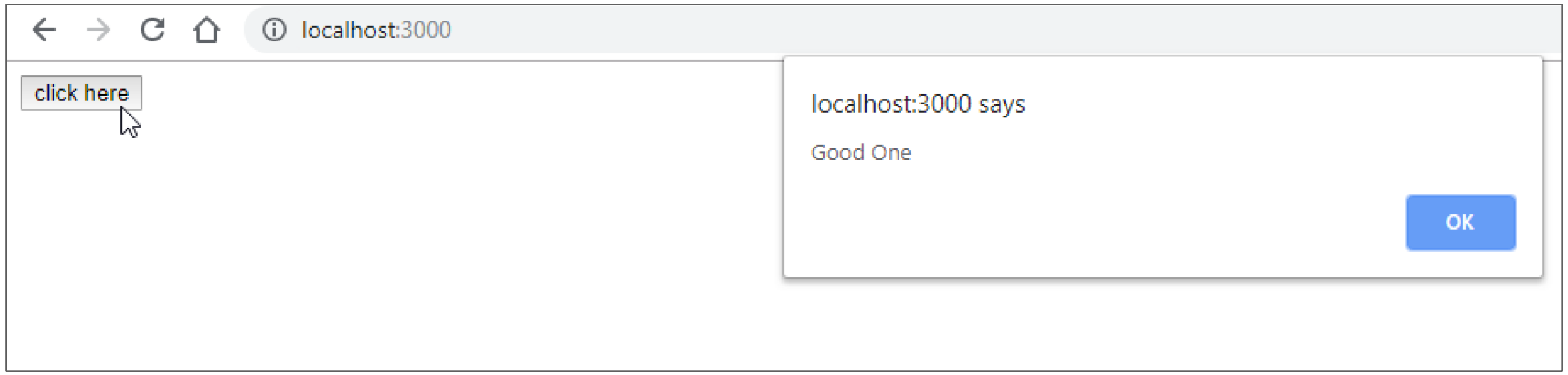
Event component

Event handler

Event

Text on button

Output:



localhost:3000 says

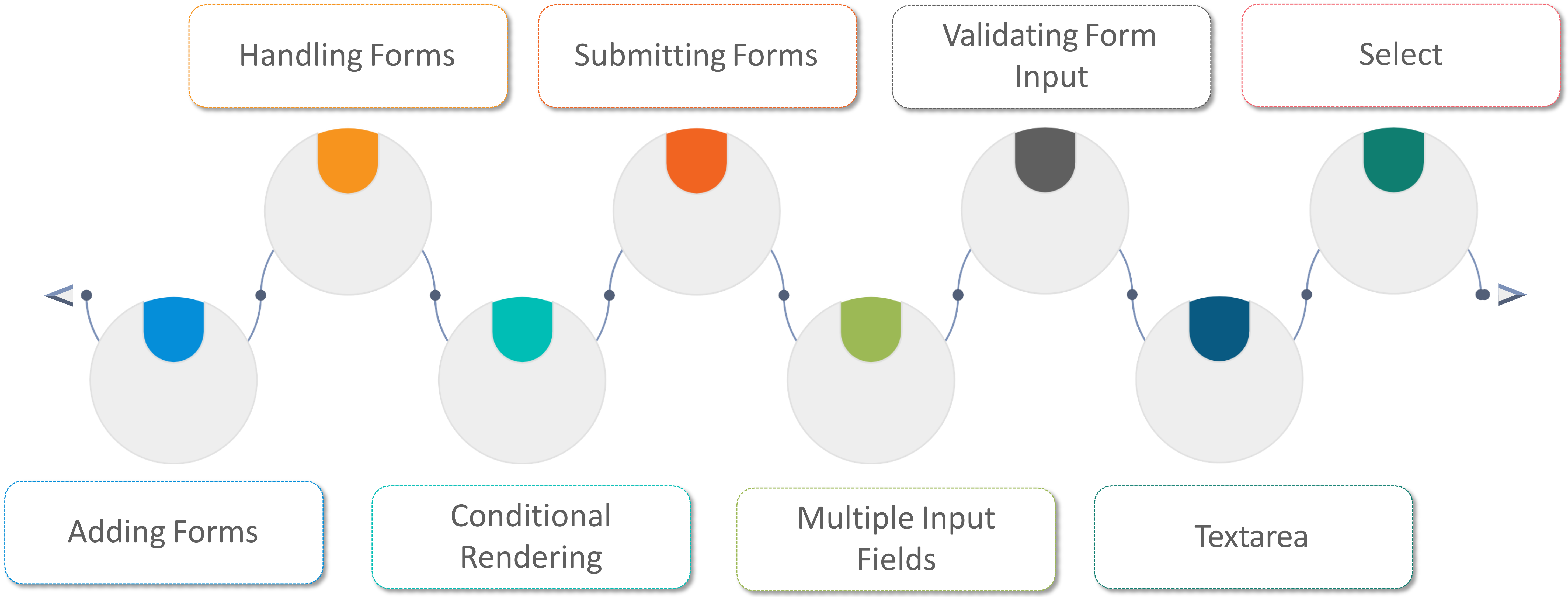
Good One

React Forms

React Forms

React Forms are designed to let users interact with a ***Web Page***

Different activities associated with React forms are:



Adding Form

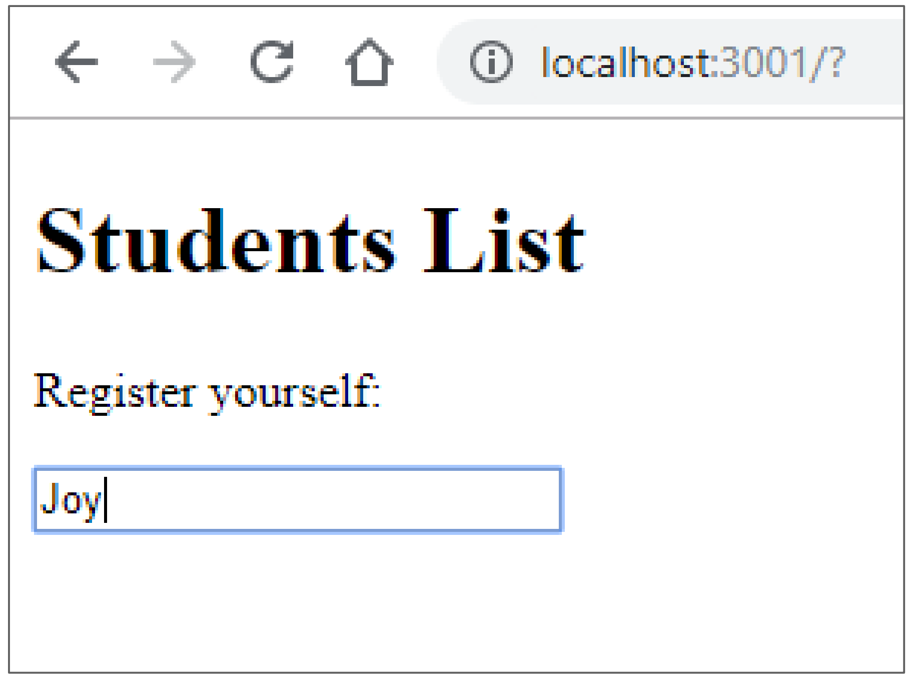
Below is the example of form which accepts the user inputs.

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

class Form extends Component {
  render() {
    return (
      <form>
        <h1>Students List</h1>
        <p>Register yourself:</p>
        <input
          type="text"
        />
      </form>
    );
  }
}
ReactDOM.render(<Form />, document.getElementById('root'));
```

HTML element to create form

Input field where the user can enter data

Output



Students List

Register yourself:

Joy

Handling Forms

Handling forms refers to managing the data on submission or when the values are changed.

```
class Form extends Component {
 constructor() {
  super();
  this.state = { participate: '' };
 }
 changeHandler = (event) => {
  this.setState({participate:  event.target.value});
 }
 render() {
  return (
   <form>
   <h1>Welcome {this.state.participate}</h1>
   <p>Register your name:</p>
   <input
    type='text'
    onChange={this.changeHandler}
   />
   </form>
  );}}
ReactDOM.render(<Form />, document.getElementById('root'));
```

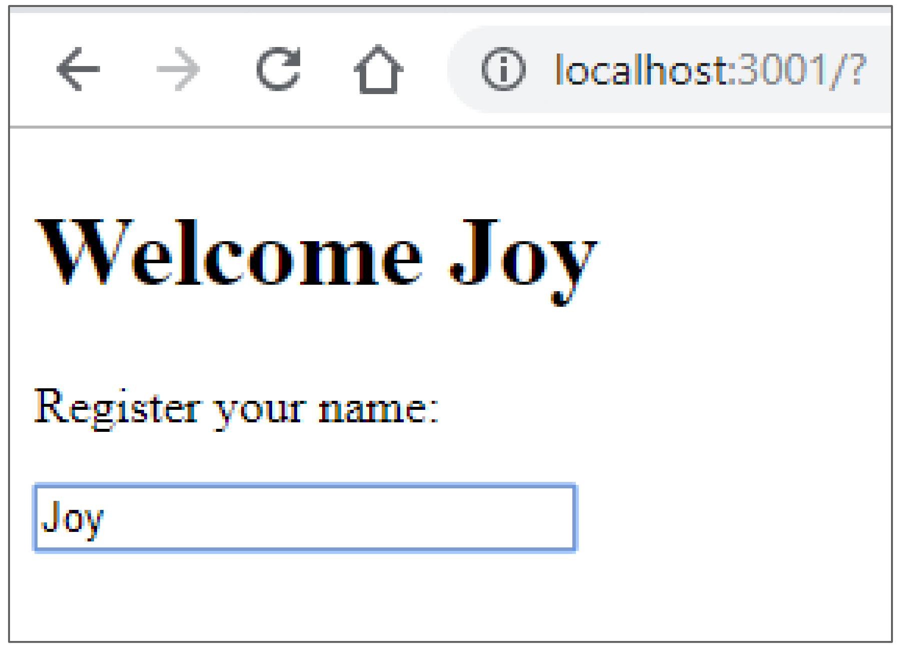
State storing the data

Records the changes

Updated or submitted value

- React form is maintained by the React components and stored in component state
- These changes can be controlled by the addition of ***onChange*** attribute

Output



Welcome Joy

Register your name:

Joy

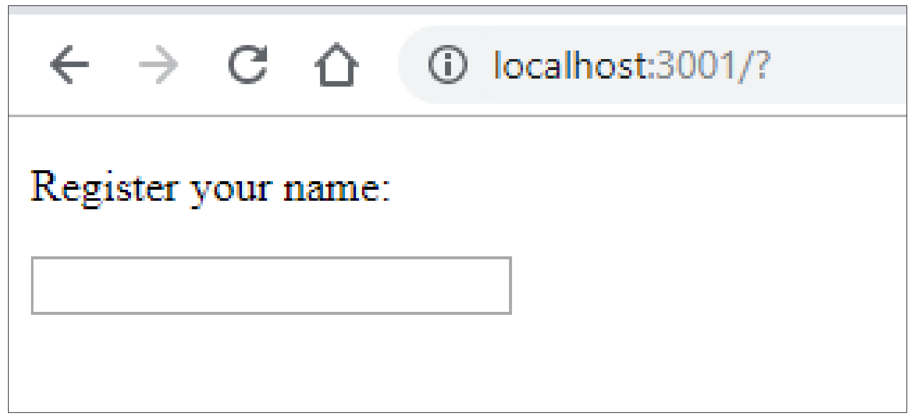
Conditional Rendering

```
class Form extends Component {
 constructor(props) {
 super(props);
  this.state = { participate: '' };
 }
 changeHandler = (event) => {
  this.setState({participate: event.target.value});
 }
 render() {
  let header = '';
  if (this.state.participate) {
   header = <h1>Thank you for Registration {this.state.participate}</h1>;
  }
  return (
   <form>
   {header}
   <p>Register your name:</p>
   <input
    type='text'
    onChange={this.changeHandler}
   />
   </form>
  );}}
ReactDOM.render(<Form />, document.getElementById('root'));
```

Conditional Rendering is usually preferred to display the data after user interaction (submission).

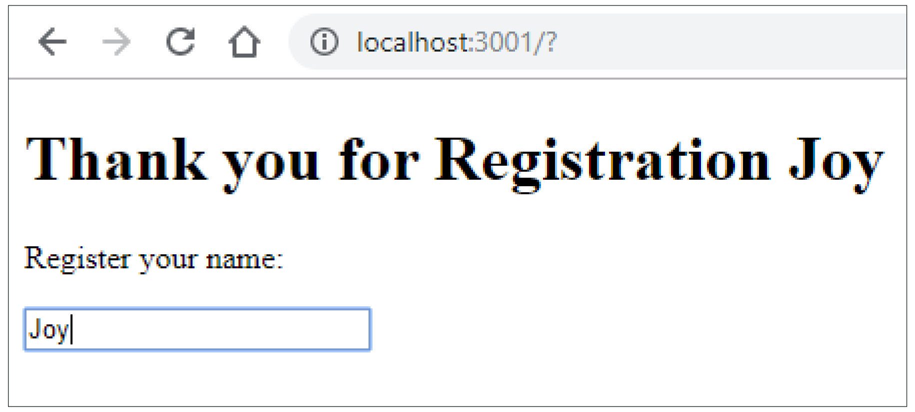
Condition to ***render Header*** after participate registration

Output



localhost:3001/?

Register your name:



localhost:3001/?

Thank you for Registration Joy

Register your name:

Joy

Forms Submission

```
class Form extends Component {
  constructor() {
    super();
    this.state = { participate: '' };
  }
  submitHandler = (event) => {
    event.preventDefault();
    alert(this.state.participate + " Registered" );
  }
  changeHandler = (event) => {
    this.setState({participate: event.target.value});
  }
  render() {
    return (
      <form onSubmit={this.submitHandler}>
      <h1>Welcome</h1>
      <p>Register your name and click on submit:</p>
      <input
        type='text'
        onChange={this.changeHandler}
      />
      <input
        type='submit'
      />
      </form>
    );}}
ReactDOM.render(<Form />, document.getElementById('root'))
```

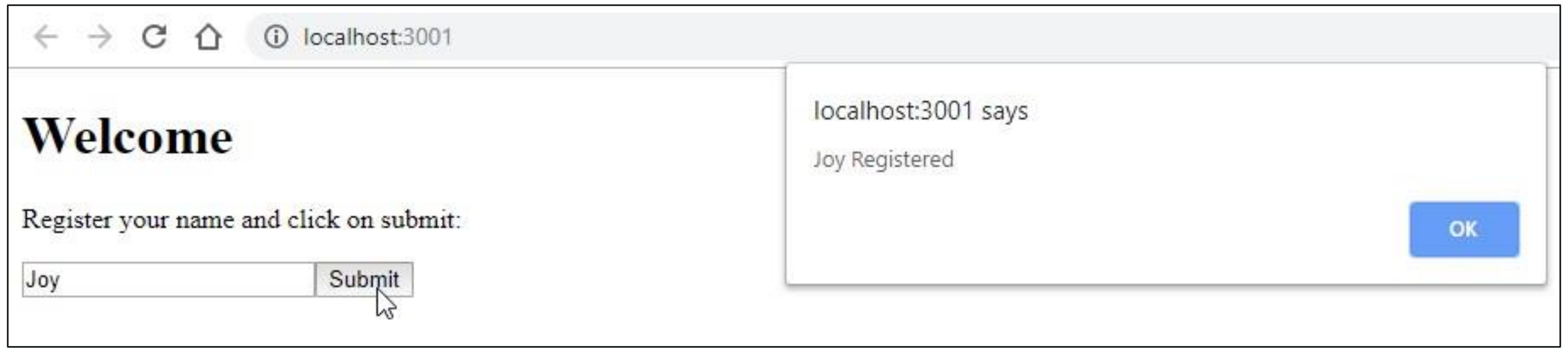
Form Submission refers to submission of data with user confirmation by clicking the submit button.

Event to be submitted after clicking submit button

Submits the entered data

Defines a submit button which submits all form values to a form-handler

Output


localhost:3001

Welcome

Register your name and click on submit:

Joy Submit

localhost:3001 says

Joy Registered

OK

Multiple Input Fields

```
class Form extends Component {
  constructor() {
    super();
    this.state = {
      participate: '',
      roll_no: null,
    };}
  changeHandler = (event) => {
    let nam = event.target.name;
    let val = event.target.value;
    this.setState({[nam]: val});
  }
  render() {
    return (
      <form>
      <h1>Hello {this.state.participate} </h1>
      <p>Register your name:</p>
      <input
        type='text'
        name='participate'
        onChange={this.changeHandler}
      />
      <p>Enter your roll_no:</p>
      <input
        type='text'
        name='roll_no'
        onChange={this.changeHandler}
      />
      <h2>Your roll_no is {this.state.roll_no}</h2
>
      </form>
    );}}
ReactDOM.render(<Form/>,
document.getElementById(' root'));
```

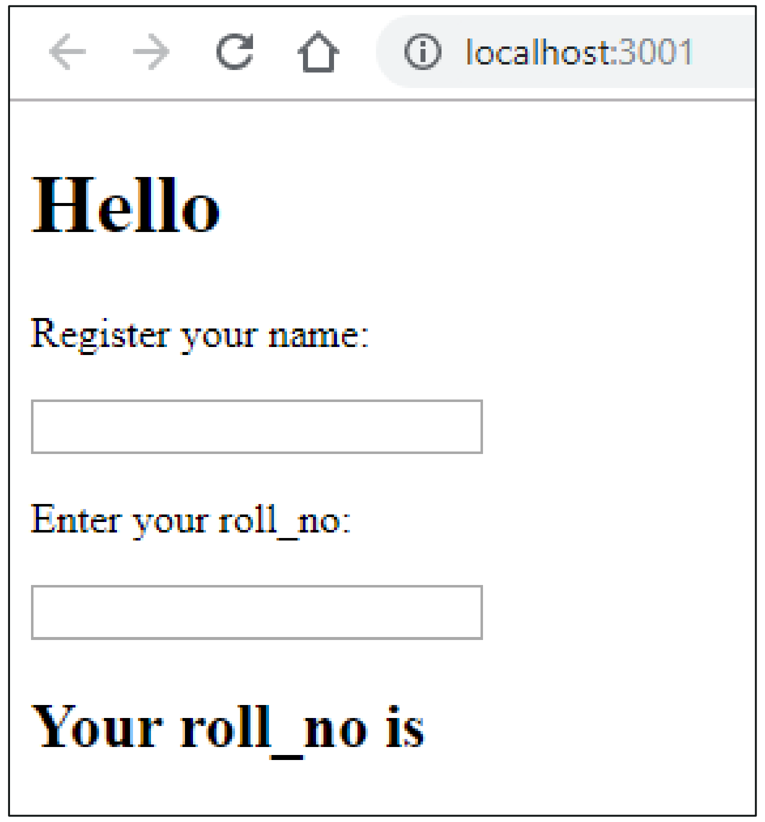
Multiple input fields include different categories to be mentioned in the form.

Manages the updated values of name and roll_no

Collects the username

Collects the user roll_no

Output

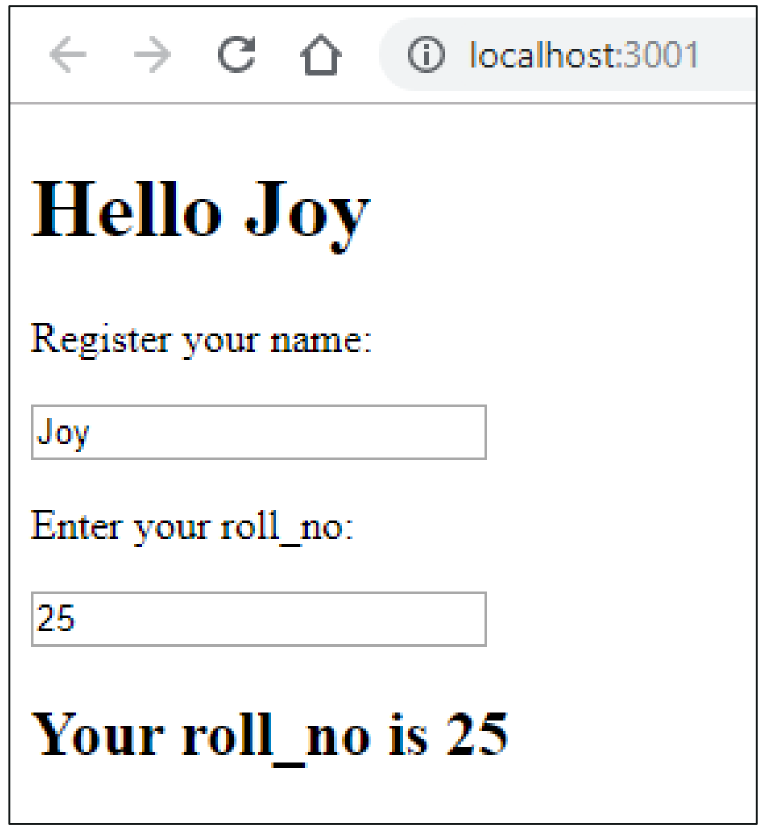

localhost:3001

Hello

Register your name:

Enter your roll_no:

Your roll_no is


localhost:3001

Hello Joy

Register your name:

Joy

Enter your roll_no:

25

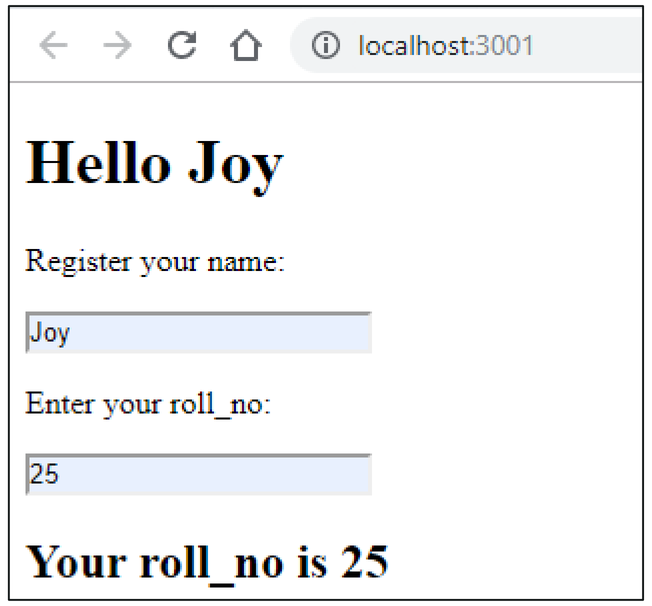
Your roll_no is 25

Validating Form Input

```
class Form extends Component {
  constructor() {
    super();
    this.state = {
      participate: '',
      roll_no: null,
    };}
  changeHandler = (event) => {
    let nam = event.target.name;
    let val = event.target.value;






  render() {
    return (
      <form>
      <h1>Hello {this.state.participate} </h1>
      <p>Register your name:</p>
      <input
        type='text'
        name='participate'
        onChange={this.changeHandler}
      />
      <p>Enter your roll_no:</p>
      <input
        type='text'
        name='roll_no'
        onChange={this.changeHandler}
      />
      <h2>Your roll_no is {this.state.roll_no}</h2>
      </form>
    );}}
ReactDOM.render(<Form />, document.getElementById('root
'))
```

Form validation refers to entering the right input, if user enters some wrong values then the input is not accepted.

Output: When you enter right data



localhost:3001

Hello Joy

Register your name:

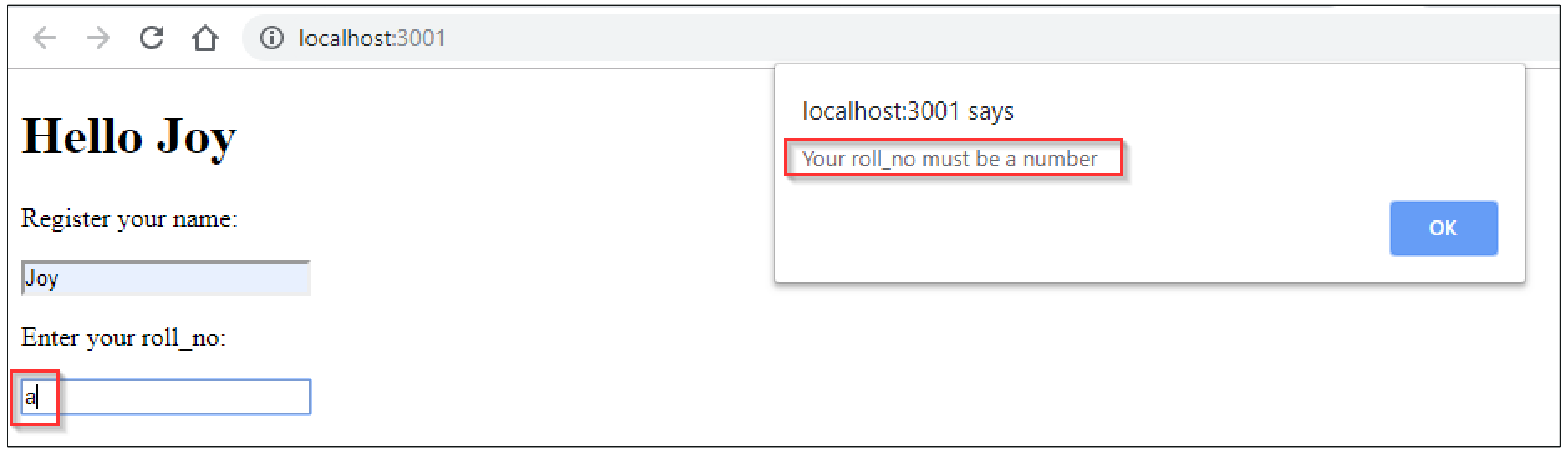
Joy

Enter your roll_no:

25

Your roll_no is 25

Output: When you enter wrong data



localhost:3001

Hello Joy

Register your name:

Joy

Enter your roll_no:

a

localhost:3001 says

Your roll_no must be a number

Instead of alert box the more preferred way is ***displaying the error when the user enters invalid data***

Refer M2 Demo Document 3 on LMS to display errors directly instead of alert box

Textarea

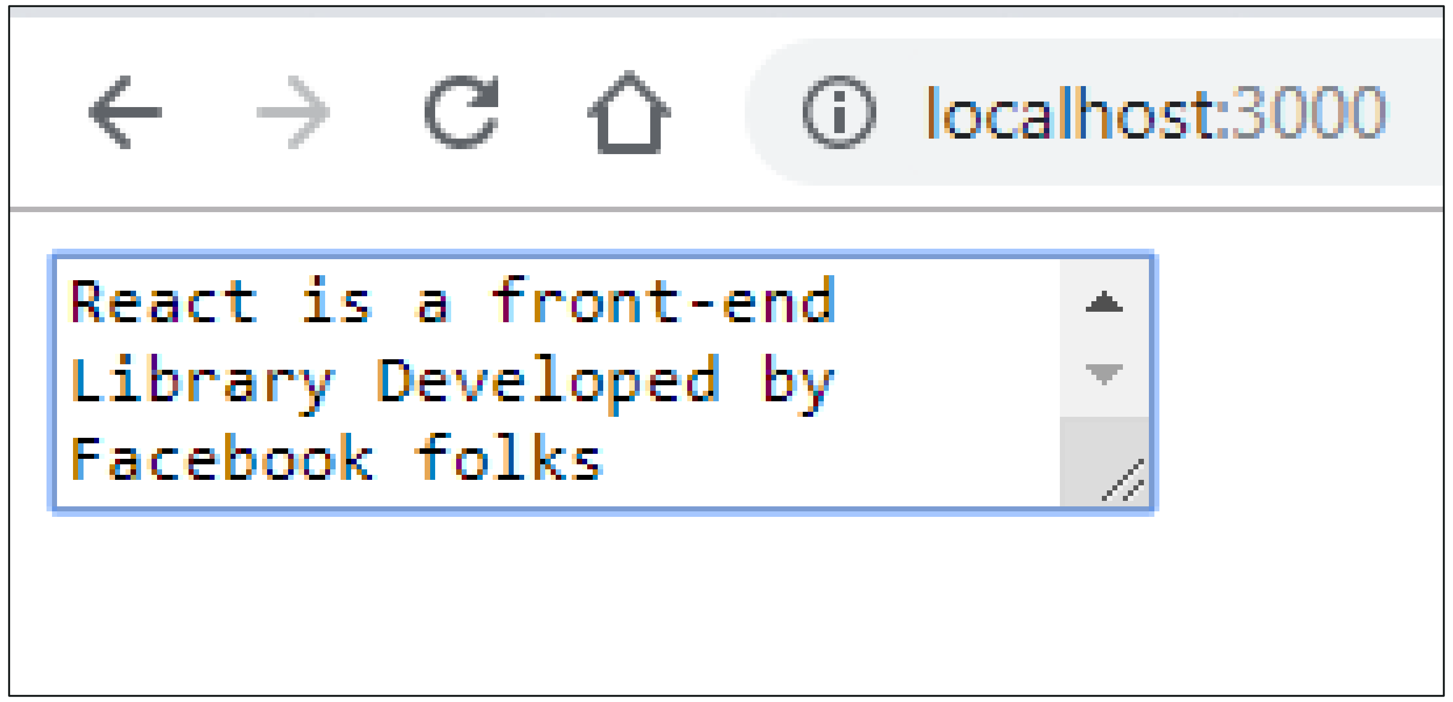
Textarea is one of the features of form, where data can be entered in textbox.

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

class Form extends Component {
  constructor() {
    super();
    this.state = {
      description: 'React is a front-
end Library Developed by Facebook folks'
    };
  }
  render() {
    return (
      <form>

      </form>
    );}}
ReactDOM.render(<Form />, document.getElementById('root'));
```

Note

In React the value of a textarea is placed in a ***value attribute***

Output



localhost:3000

```
React is a front-end
Library Developed by
Facebook folks
```

Select

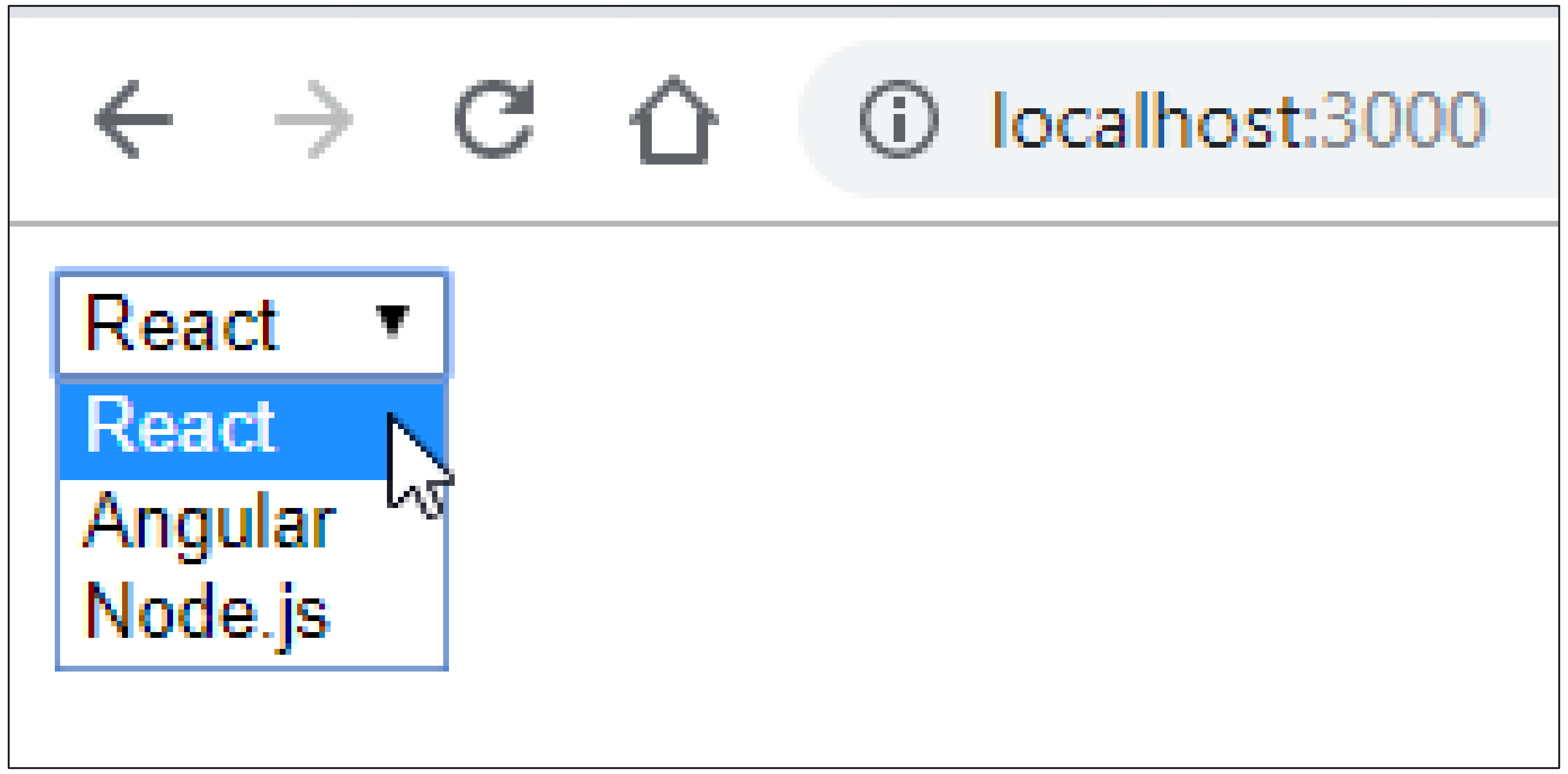
Select feature offers list of options, where user is supposed to make a choice of appropriate option.

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

class Form extends Component {
  constructor() {
    super();
    this.state = {
      myTraining: "choose"
    };
  }
  render() {
    return (
      <form>






      </form>
    );}}
ReactDOM.render(<Form />, document.getElementById('root'));
```

Note

In React, the selected value is defined with a value attribute on the select tag

Output



Styling In React

Here we will learn how to improve our application representation using the ***CSS.***

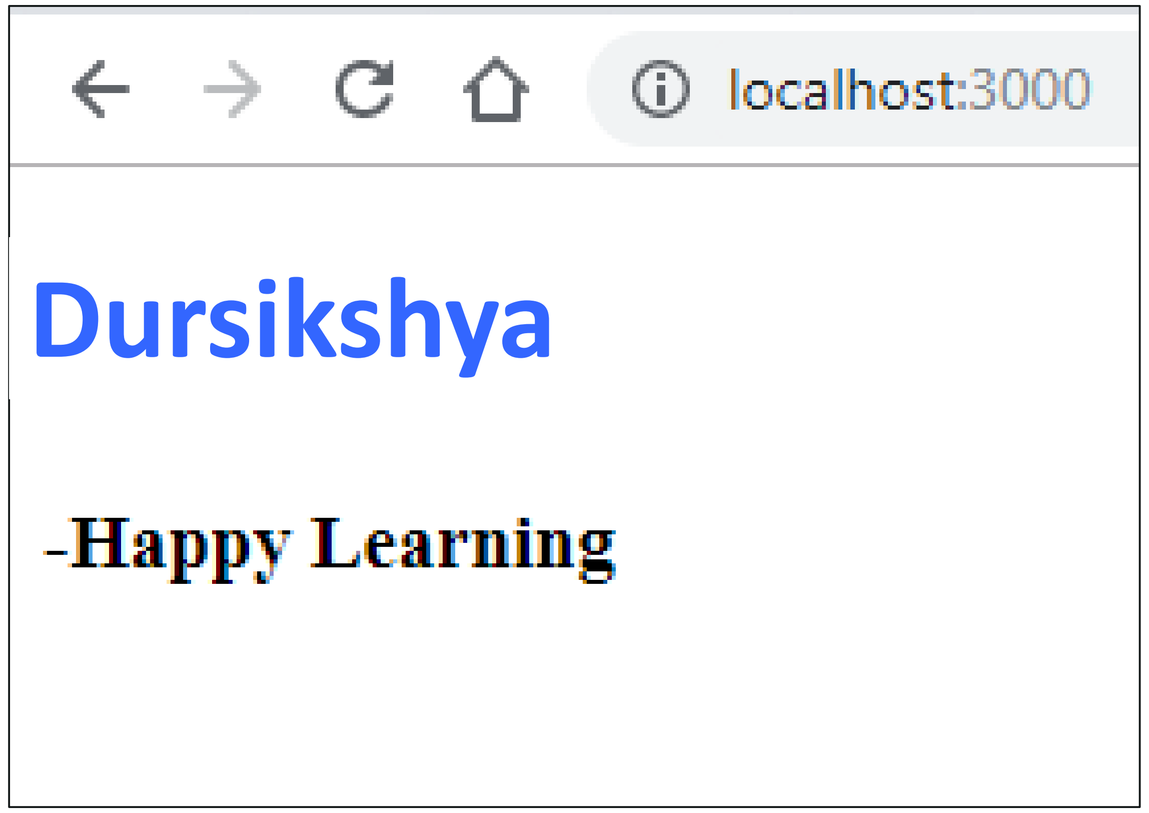
Inline Styling

Inlining CSS means putting your **CSS** into your HTML file instead of an external **CSS** file.

To style an element with the inline style attribute, the value must be a JavaScript object

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

class Title extends Component {
  render() {
    return (
      <div>


      </div>
    );
  }
}

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

Output


localhost:3000

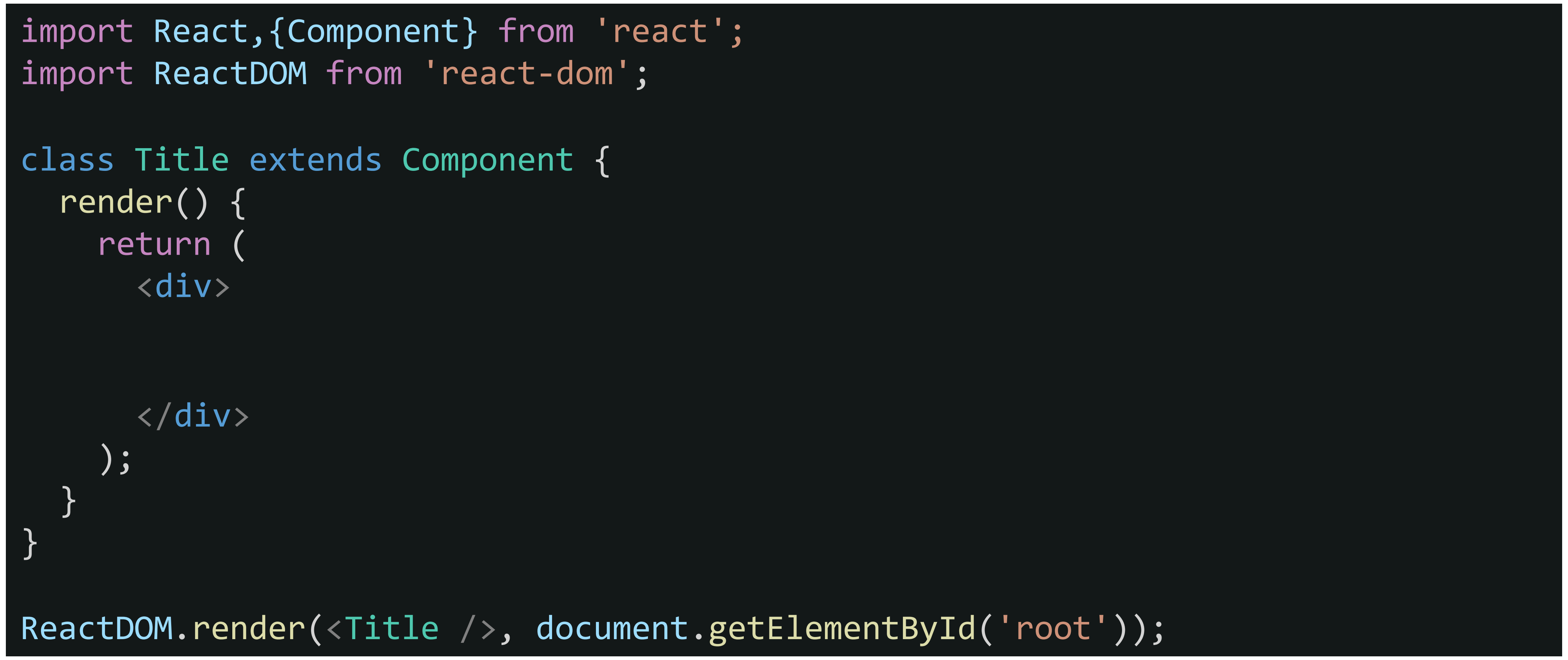
Dursikshya

-Happy Learning

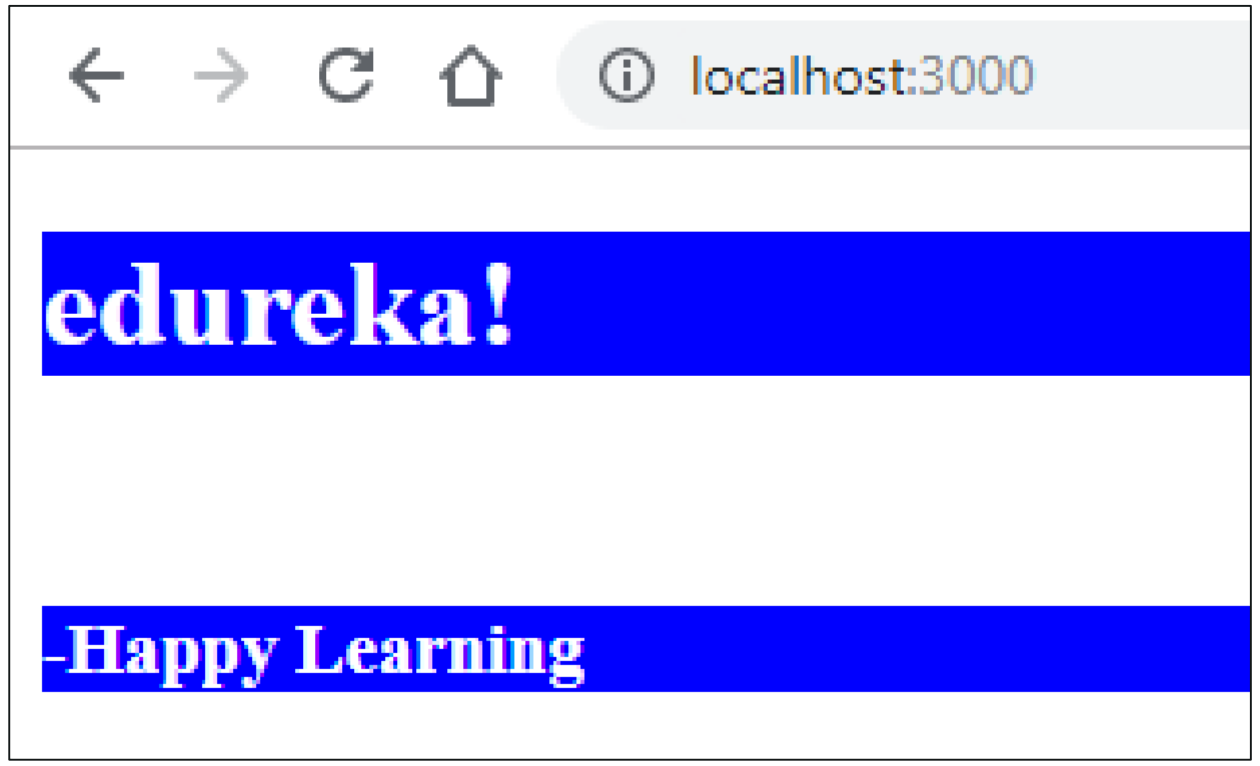
In JSX, JavaScript expressions are written inside curly braces, and since JavaScript objects also use curly braces, the styling in the example above is written inside two sets of ***curly braces {{}}***

Adding Background Color To Text

In Inline CSS, properties with ***two words*** like ***background-color***, must be written in camelCase syntax.



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

class Title extends Component {
  render() {
    return (
      <div>


      </div>
    );
  }
}

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



localhost:3000

edureka!

-Happy Learning

CSS Stylesheet

This is an another way where CSS styling is written in a separate file and saved with the ***.css file extension***, which later you can import it in your application.

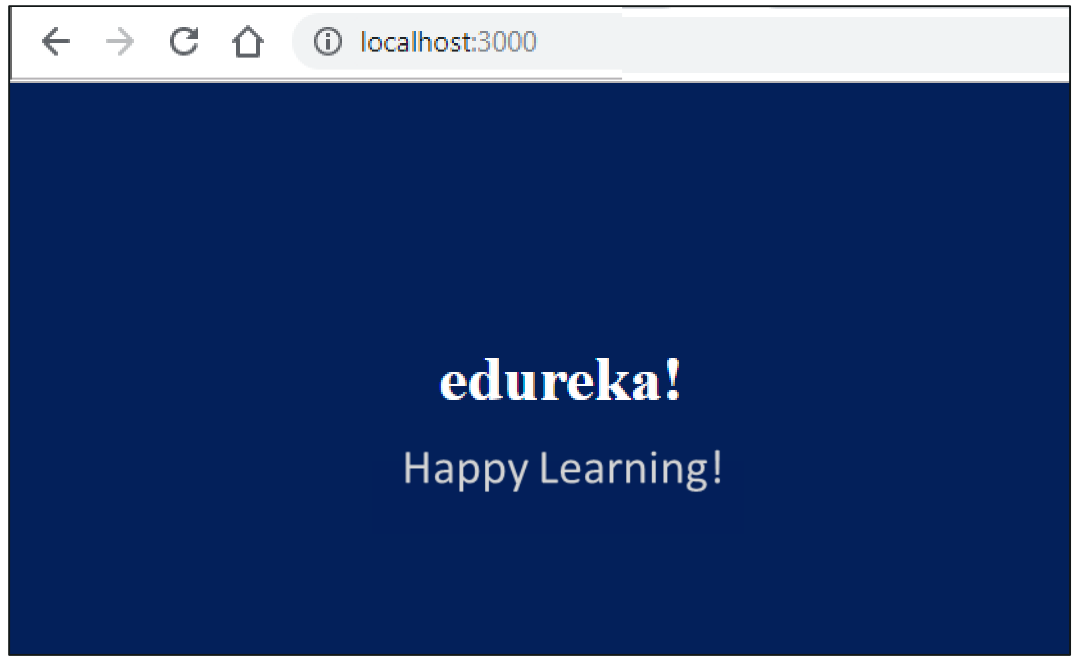
App.css

```
body {
    background-color: #03205a;
    color: rgb(255, 255, 255);
    padding: 100px;
    font-family: 'Gill Sans';
    text-align: center;
  }
```

Index.js

```
import React,{Component} from 'react';
import ReactDOM from 'react-dom';
import './App.css';

class Title extends Component {
  render() {
    return (
      <div>
      <h1>edureka!</h1>
      <p>Happy Learning!</p>
      </div>
    );
  }
}

ReactDOM.render(<Title />, document.getElementById('r
oot'));
```

Output



edureka!

Happy Learning!

Demo 4: Build A Music Store Application Using React Components

Questions

FEEDBACK

Comments

Ratings

Suggestions

Survey

Likes

Ideas