State in React Components

Topics covered:

- What is the state of component?
- How to use state in class based components?
  - ➔ Syntax of setState
  - ➔ Use function to update state
- How to use state in functional component?
  - ➔ useState Hooks
- Difference between state and props.

1. **State of component**:
   - State is the most complicated aspect of React, and it is something that both new and veteran developers struggle with. We will therefore examine all the fundamentals of state in React.
   - A state object is already present in React components.
   - Property values for the component's properties are kept in the state object.
   - The component re-renders whenever the state object alters.

   Example:

   The constructor initializes the state object:

   ```
   this.state = {name: "sony",rollno: "202126" };
   ```

   Add a constructor method parameter specifying the state object:

```
export class Details extends Component {
  constructor(props) {
    super(props);
    this.state = { name: "Avni",rollno: "202126"};
  }
render() {
    return (
          <div>
             <p>
              Student Name: {this.state.name} <br /> Roll no:
              {this.state.rollno}
             </p>
           </div>
    );
  }
```

```
}
```

Output:

Student Name: Avni
Roll no:202126

2. **Use state in class-based component:**
   - An internal state object exists in React Class components.
   - Perhaps you noticed that we used state in the component constructor section previously.
   - The state object is where you save the component's property values.
   - The component re-renders as soon as the state object is modified.

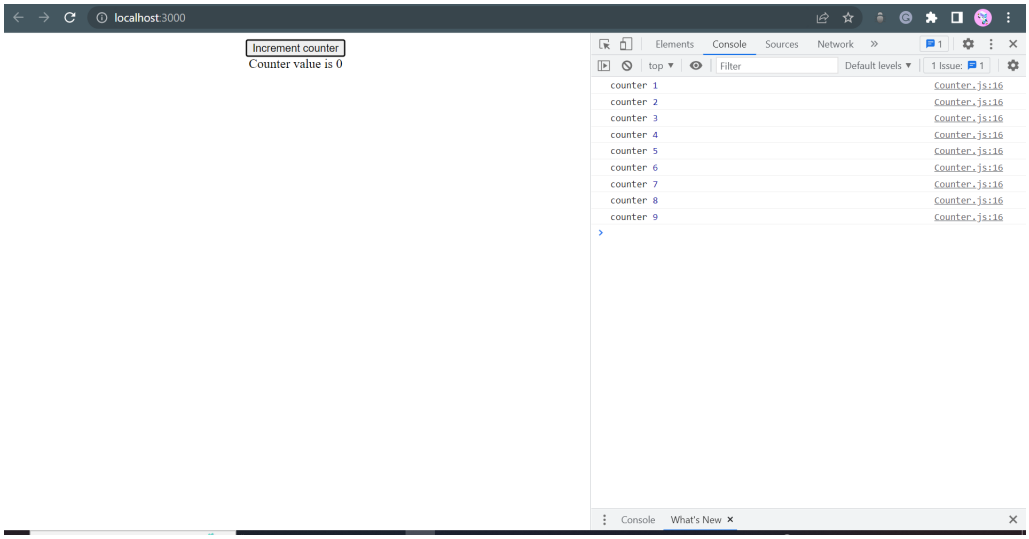
Example:

```
import React from "react";

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

   this.state = {
     counter: 0
   };

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

 handleClick() {
     this.state.counter = this.state.counter + 1;

     console.log("counter", this.state.counter);
 }

 render() {
   const { counter } = this.state;
```

```
    return (
      <div>
        <button onClick={this.handleClick}>
         Increment counter</button>
        <div>Counter value is {counter}</div>
      </div>
    );
  }
}

const rootElement = document.getElementById("root");
ReactDOM.render(<Counter />, rootElement);
```

Examine what we're doing here:

- We first call super inside the constructor function and pass it props. After that, we defined the state as an object with the property of counter.
- In order to ensure that the handleClick method receives the appropriate context for this, we are also binding this's context to it.
- The counter is then updated and logged to the console inside the handleClick method.
- We also return the JSX that we wish to render on the UI inside the render method.



As you can see in the console, the counter is correctly updating, but it is not updating in the user interface.
This is because, inside the handleClick method, we are immediately updating the state as:

```
this.state.counter = this.state.counter + 1;
```

React does not re-render the component as a result (and explicitly updating state is likewise bad practice).

Directly updating or changing state in React is never a good idea because it will break your application. Additionally, if you make a direct state change, your component won't be re-rendered on state change.

Syntax of setState

- React provides us with a setState function that enables us to modify the state's value in order to make a change.

The syntax for the setState function is as follows:

```
setState(updater, [callback])
```

- *Updater* may be an object or a function.
- The optional function *callback* is executed after the state has been successfully modified.

Note: The component and all of its children are automatically rerendered when setState is called on them. As previously demonstrated with the renderContent function, we don't need to manually render again.

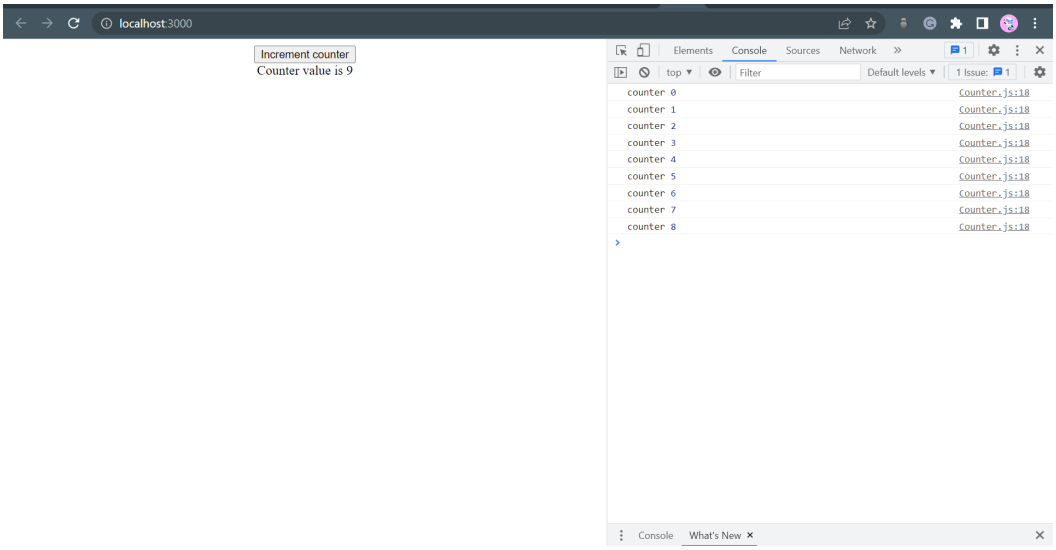
Use function to update state:

- The modified handleClick function appears to be as follows:

```
handleClick() {
  this.setState((pre) => {
    return { counter: pre.counter + 1 };
  });
  console.log("counter", this.state.counter);
}
```

In this case, we send a function as the first argument to the setState function, and we return a new state object with counter incremented by one based on the previous value of counter.

In the preceding code, we're using the arrow function, but any standard function would work.



- We're getting the updated value of the counter on the UI, as you can see. However, even if we inserted console.log after the this.setState method, we still get the previous counter value in the console.

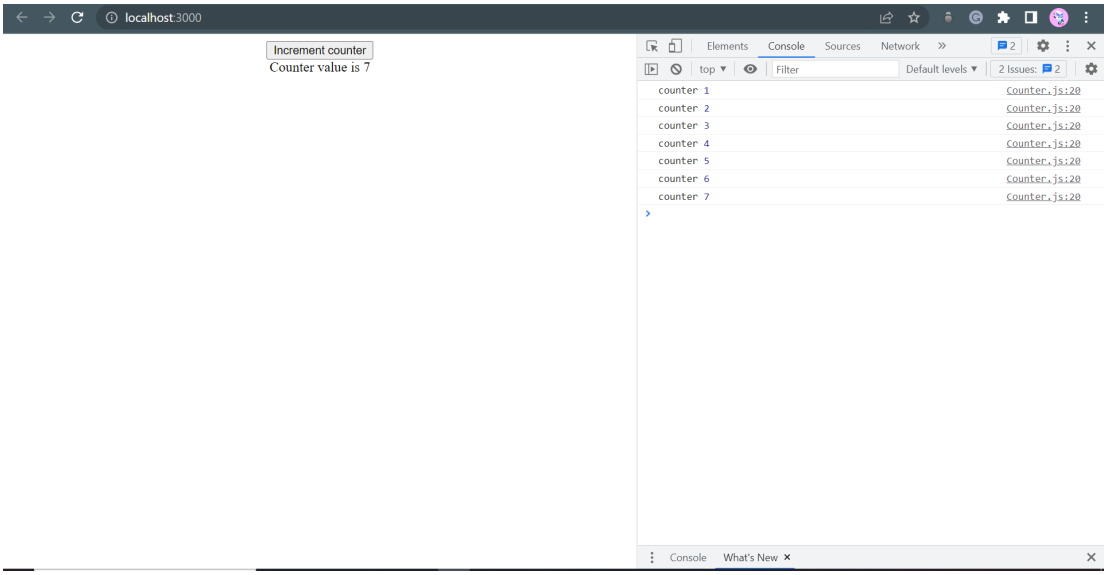
- This is due to the asynchronous nature of the setState function.

- This indicates that even though we called setState to increment the counter value by one, it did not happen right away.

- This is due to the fact that when we execute the setState function, the entire component is re-rendered - therefore React needs to examine what needs to be modified using the Virtual DOM algorithm and then do other checks for an efficient UI update.

You can supply a function as the second argument to the setState call, which will be performed once the state is modified, if you wish to quickly obtain the updated value of the state following the setState call.
Like this:

```
handleClick() {
   this.setState(
     (pre) => {
       return { counter: pre.counter + 1 };
     },
     () => {
       console.log("counter", this.state.counter);
     }
   );
 }
```



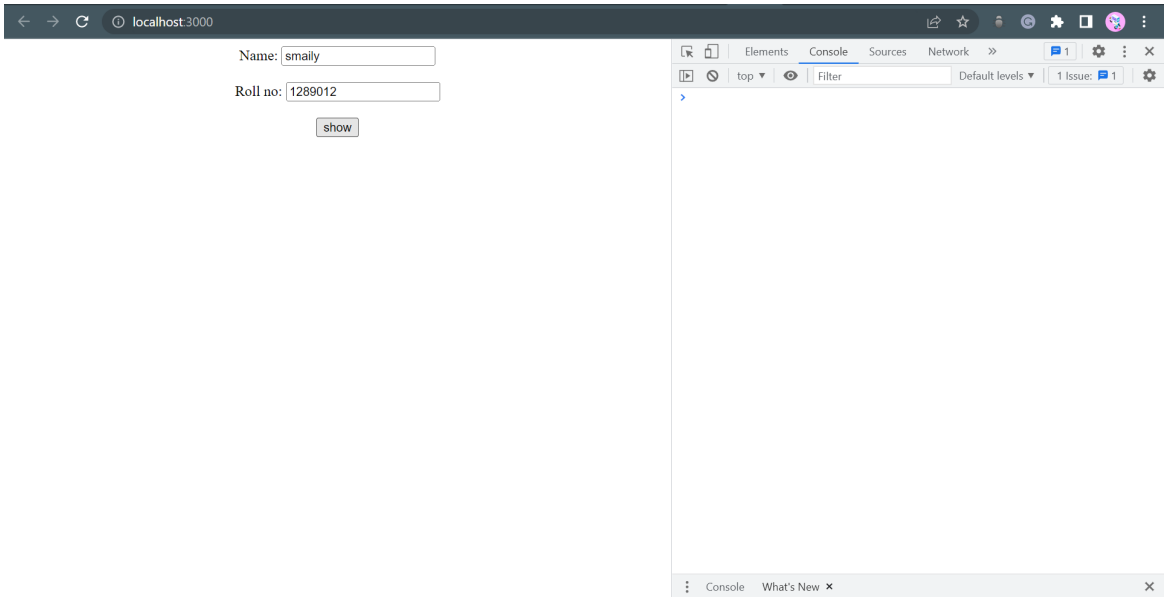
- We are sending two arguments in this case while calling the setState function. The first is a function that creates a new state, and the second is a callback function that is invoked whenever the state is altered. In the callback function, we are only logging the modified counter value to the console.

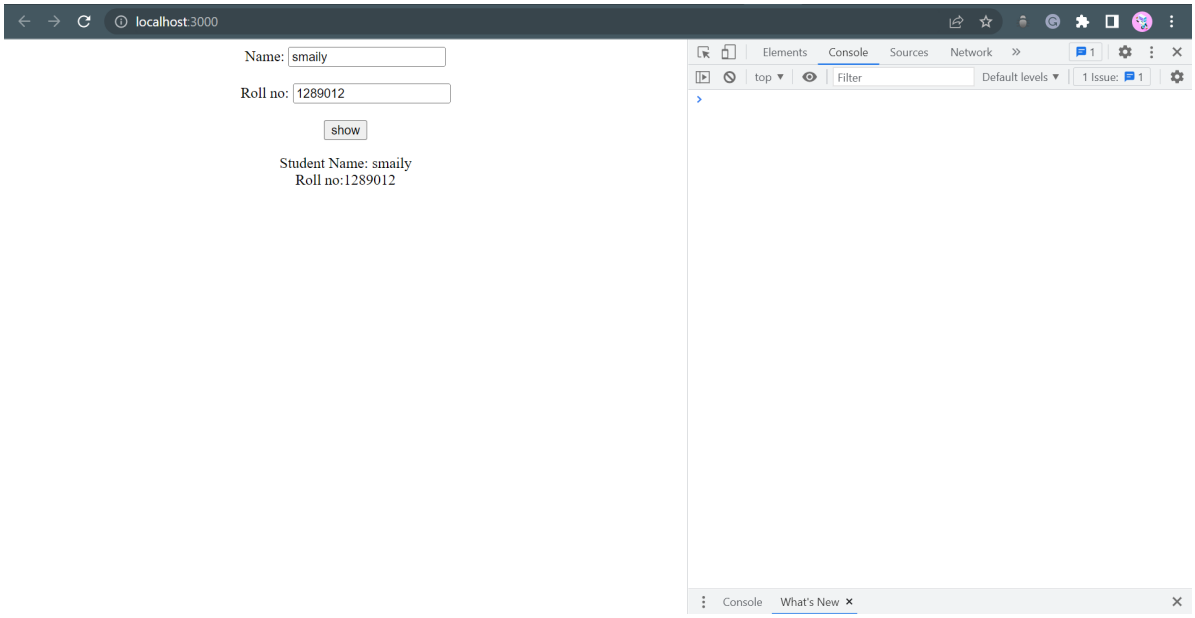
Example:

```
export class Details extends Component {
  constructor(props) {
    super(props);
    this.state = { name: "", roll_no: "", show: false };
  }
  changeHandle1 = (e) => {
    this.setState({ name: e.target.value });
  };
  changeHandle2 = (e) => {
    this.setState({ roll_no: e.target.value });
  };

  render() {
    return (
      <div>
        <label>Name: </label>
        <input
          type="text"
          value={this.state.name}
          onChange={this.changeHandle1}
        />
        <br />
        <br />
        <label>Roll no: </label>
        <input
```

```
          type="text"
          value={this.state.roll_no}
          onChange={this.changeHandle2}
        />
        <br />
        <br />
        <button
          onClick={() => {
            this.setState((pre) => {
              return { show: !pre.show };
            });
          }}
        >
          show
        </button>
        {this.state.show && (
          <p>
            Student Name: {this.state.name} <br /> Roll no:
            {this.state.roll_no}
          </p>
        )}
      </div>
    );
  }
}
```

Output:





Use state in functional component:

- With the exception of not having state and lifecycle methods, functional components are similar to class components. This is the reason they are sometimes referred to as stateless functional components.

- Hooks were added to React in version 16.8.0. Additionally, they have entirely altered the way we write code for React. We can leverage state and lifecycle methods inside functional components by using React Hooks.

- Functional components with additional state and lifecycle methods are known as react hooks.

  useState hook in react:

- We must apply the *useState* hook in order to define state using React Hooks.

- The initial value of the state is a parameter that the *useState* hook accepts.

- State is always an object in components that use classes. But when using useState, you can give any value—such as a number, string, boolean, object, array, null, and so on—as the initial value.

- The first value of the array that the useState hook delivers is the state's current value. The function we will use to update the state in a manner similar to the setState method is represented by the second value.

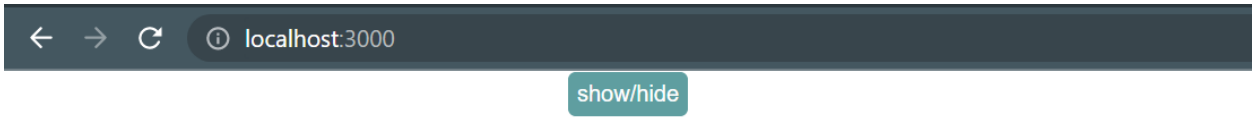
Let's look at a class-based component that uses the state object. Using hooks, we'll turn it into a functional component:

```
import React from "react";
export default class ShowHide extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      show: false,
    };
  }
  showFun = () => {
    this.setState((prev) => {
      return { show: !prev.show };
    });
  };
  render() {
    return (
      <div>
        {this.state.show && <div className="block">block</div>}
        <button onClick={this.showFun}>show/hide</button>
      </div>
    );
  }
}
```

Lets convert the above code in functional useState hooks:

```
import React, { useState } from "react";
export default function ShowHide() {
  const [show, setShow] = useState(false);

  return (
    <div>
      {show && <div className="block">block</div>}
      <button
        onClick={() => {
          setShow((prev) => !prev);
        }}
      >
        show/hide
      </button>
    </div>
  );
}
```

Output:



Let's examine the code shown above.

- We must import the *useState* hook like we did in the first line in order to use it.
- Within the App component, we are utilizing destructuring syntax and supplying *false* as the starting value when we use useState. The *show* and *setShow* variables were used to store the array values returned by useState.
- Prefixing the name of the function used to update the state with the set keyword, as in *setShow*, is a typical practice.
- The *setShow* function is called by defining an inline function and sending the updated *Show* value when the increment button is pressed.

Note that we used the sowh value, which we already had, to change the show (boolean value) by using the formula setShow((pre)=>!pre).
There is no need to relocate the code into a different function because the inline on click handler only contains one statement. However, you have the option to do that if the handler's code becomes complicated.

Difference between state and props:

Let's review and examine the key distinctions between props and state, then:

- While state allows components to create and maintain their own data, props allow them to receive data from outside the component.
- Data is passed using props, whereas state is used to manage data.
- Data from props is read-only and cannot be changed by an external component sending it.
- State data is private but can be altered by its own component (cannot be accessed from outside).
- The only way to transmit a prop is from a parent component to a child (unidirectional flow).
- SetState () method should be used to modify state.