Component States and Life Cycle Methods

Introduction to Component States

The state is an instance of React Component Class can be defined as an object of a set of observable properties that control the behavior of the component. In other words, the State of a component is an object that holds some information that may change over the lifetime of the component. t **stores a component's dynamic data** and determines the component's behaviour. Because state is dynamic, it enables a component to keep track of changing information in between renders and for it to be dynamic and interactive.

Defining the state

State can only be accessed and modified inside the component and directly by the component. Must first have some **initial state**, should define the state in the constructor of the component's class.

```
class MyClass extends React.Component {
        constructor(props)  {
        super(props);  this.state = { attribute : "value" };    }
}
```

The state object can contain as many properties as you like

```
class MyClass extends React.Component {
        constructor(props)  {
        super(props);   this.state = { attribute1 : "value1", attribute2 : "value2",
attribute3 : "value3"};    }            }
```

Changing the state

State should **never be updated explicitly**. Must use **setstate().** It takes a single parameter and expects an object which should contain the set of values to be updated. Once the update is done the method implicitly calls the render() method to repaint the page.

```
this.setState({attributen : "valuen", attribute2 : "newvalue1", })
```

Differences between state and props

- States are mutable and Props are immutable
- States can be used in Class Components, Functional components with the use of React Hooks (useState and other methods) while Props don't have this limitation.
- Props are set by the parent component. State is generally updated by event handlers

Example code 1: Changing the state based on the click on the button

```
<body>
<div id = "root"> </div>
<script type = "text/babel">
    class Hello extends React.Component {
        constructor(props)      {
                super(props)
                this.state = {name:"sindhu",address:"nagarbavi",phno:"9876554"     }
                //this.updatestate = this.updatestate.bind(this) // if the function has no =>
        }
        render(){
                return (<div><h1>hello {this.state.name}</h1>
                <p> u r from {this.state.address} and ua contact number is
                {this.state.phno}</p>
                <button onClick = {this.updatestate}>click here to change state</button>
                </div> )
        }
        updatestate =()=>       {
                this.setState({name:"pai"})         }
    }
  ReactDOM.render(<Hello/>, document.getElementById("root"))
</script> </body>
```

Example code 2: Generate the Digital Clock using ReactJS

```
<script type = "text/babel">
        class Clock extends React.Component{
        constructor(){
                super()
                this.state = {time: new Date()}
                //this.tick = this.tick.bind(this)
        }
        //tick()
        tick = () => {          this.setState({time:new Date()})          }
        render(){   setInterval(this.tick,1000)
                    return (<h1>{this.state.time.toLocaleTimeString()}</h1>)
        }
       }
        ReactDOM.render(<Clock/>, document.getElementById("root"))
</script>
```

toLocaleTimeString(): Returns the time portion of a Date object as a string, using locale conventions.

Few points to think:

- What happens if you use setInterval directly inside class outside all functions?
- Calling the setInterval in render() is not a good idea as conventionally render function is used only to render the component . Then which is the best place to have this setInterval function call?

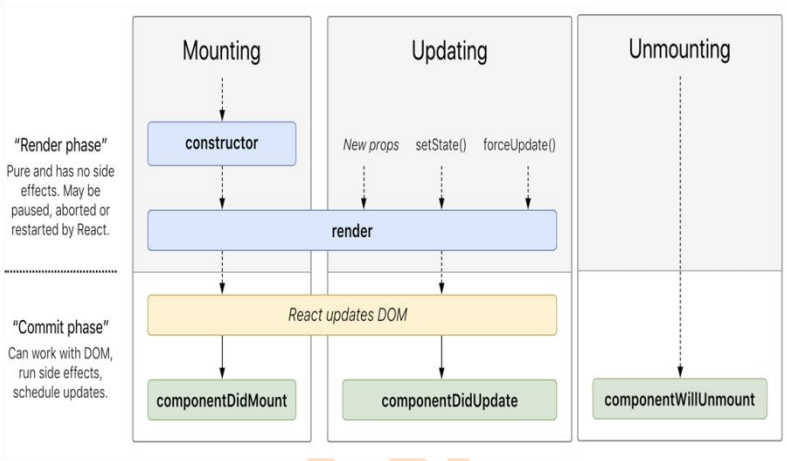
Life Cycle Methods

The series of events that happen from the starting of a React component to its ending. Every component in React should go through the following lifecycle of events.

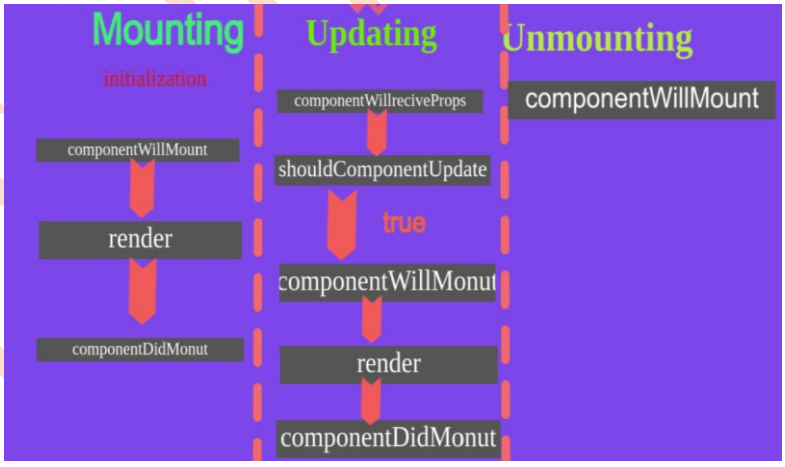
Mounting - Birth of the Component

Updating- Growing of component

Unmounting- End of the component



Functions defined in every phase is more clear with the below diagram



Functions/Methods in detail

1. constructor() : Premounting

This function is called before the component is mounted. Implementation requires calling of super() so that we can execute the constructor function that is inherited from

React.Component while adding our own functionality. Supports Initializing the state and binding our component

```
        constructor() {
  super()
  this.state = {
    key: "value"
  }
}
```

It is possible to use the constructor to set an initial state that is dependent upon props. Otherwise, this.props it will be undefined in the constructor, which can lead to a major error in the application.

```
constructor(props) {
  super(props);
  this.state = {
    color: props.initialColor
  };
}
```

2. componentWillMount()

This is called only once in the component lifecycle, immediately before the component is rendered. Executed before rendering, on both the server and the client side. Suppose you want to keep the time and date of when the component was created in your component state, you could set this up in componentWillMount.

```
componentWillMount() {
  this.setState({ startDateTime: new Date(Date.now())});
}
```

3. render()

Most useful life cycle method as it is the only method that is required. Handles the rendering of component while accessing this.state and this.props

4. **componentDidMount()**

Function is called once only, but immediately after the render() method has taken place. That means that the HTML for the React component has been rendered into the DOM and can be accessed if necessary. This method is used to perform any DOM manipulation of data-fetching that the component might need.

The best place to initiate API calls in order to fetch data from remote servers. Use setState which will cause another rendering but It will happen before the browser updates the UI. This is to ensure that the user won't see the intermediate state. AJAX requests and DOM or state updates should occur here. Also used for integration with other JavaScript frameworks like Node.js and any functions with late execution such as **setTimeout** or **setInterva**l

5. componentWillReceiveProps()

Allows us to match the incoming props against our current props and make logical. We get our current props by calling this.props and the new value is the nextProps argument passed to the method. It is invoked as soon as the props are updated before another render method is called.

6. **shouldComponentUpdate()**

Allows a component to exit the Update life cycle if there's no reason to use a replacement render. It may be a no-op that returns true. Means while updating the component, we'll re-render.

7. componentWillUpdate()

Called just before the rendering

8. componentDidUpdate()

Is invoked immediately after updating occurs. Not called for the initial render. Will not be invoked if **shouldcomponentUpdate() returns false**.

9. **componentWillUnmount()**

The last function to be called immediately before the component is removed from the DOM. It is generally used to perform clean-up for any DOM-elements or timers created in componentWillMount.

```
componentWillUnmount() {
  clearInterval(this.interval);
}
```

Example code 3: Sequence of execution of life cycle methods

```
<script type = "text/babel">
        class Hello extends React.Component{
                constructor(props){
                        console.log("in constructor")
                        super(props)
                        this.state={ame:"sindhu",address:"nagarbavi",phno:"9876"}
                        //this.updatestate = this.updatestate.bind(this)
                        //this.fun1 = this.fun1.bind(this)
                }
                render(){
                        console.log("in render")
                        return (<div><h1>hello {this.state.name}</h1>
                                <p> u r from {this.state.address} and ua contact number is {this.state.phno}</p>
                        <button onClick = {this.updatestate}>click here to change state</button>
                        <button onClick = {this.fun1}> click here to delete the user</button></div>
                        )
                }
                componentDidUpdate(prevProps, prevState){
                                console.log("component did update")
                }
                UNSAFE_componentWillUpdate()  {
                                console.log("will update")
                }
                componentDidMount(){
                                console.log("did mount")
                }
                UNSAFE_componentWillMount()  {
```

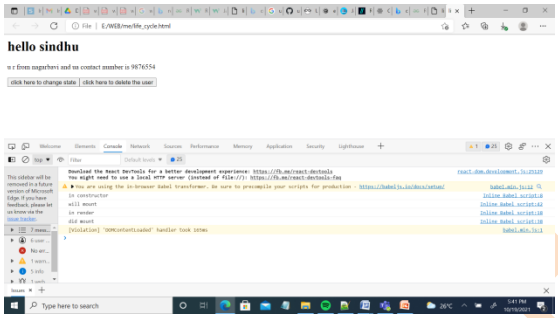
```
                    console.log("will mount")
            }
            fun1 = () =>    {
ReactDOM.unmountComponentAtNode(document.getElementById("root"))
            }
            componentWillUnmount(){
                    console.log("will unmount")
            }
            shouldComponentUpdate(nextProps, nextState) {
                console.log("yes, in shouldCOmponentupdate")
                return true;
            }
            updatestate =()=>{
                console.log("in updatestate")
                this.setState({name:"pai"})    }
            }
            ReactDOM.render(<Hello/>, document.getElementById("root"))
</script>
```

ReactDOM.unmountComponentAtNode(node to be deleted) : Used to delete the componenet from the DOM

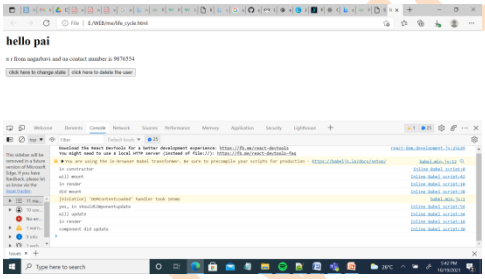
Note: In React v16.3, componentWillMount(), componentWillReceiveProps(), and componentWillUpdate() are marked as unsafe legacy lifecycle methods for deprecation process. They have often been misused and may cause more problems with the upcoming async rendering. As safer alternatives for those methods, **getSnapshotBeforeUpdate() and getDerivedStateFromProps()** were newly added. Since the roles of the unsafe methods and the newly added methods may overlap, React prevents the unsafe methods from being called when the alternatives are defined and outputs a **warning** message in the browser. So, explicitly UNSAFE_ keyword is used in redefining these methods.

Observe the output of the above code in console at every stage. Snapshot is attached in sequence.

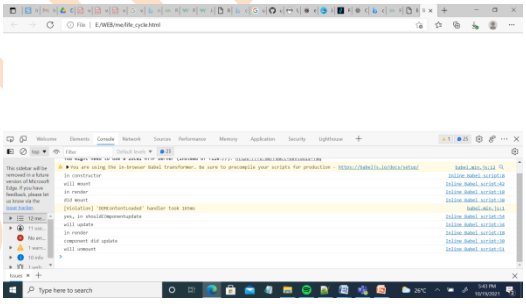
stage 1: once the page is loaded on the browser



Stage 2: when the first button is clicked



Stage 3: When the second button is clicked.



Example code 4: Digital clock using life cycle method

```
<body>
  <div id = "root"> </div>
  <script type = "text/babel">
        class Clock extends React.Component{
              constructor()   {
```

```
            super()
            this.state = {time: new Date()}
            this.tick = this.tick.bind(this)
            console.log("in constructor")
        }
        tick(){
            console.log("in tick")
            this.setState({time:new Date()})
        }
        render(){
            console.log("in render")
            return (<h1>{this.state.time.toLocaleTimeString()}</h1>)
        }
        componentDidMount(){
            console.log("in did mount")
            setInterval(this.tick,1000) // Observe that added here in didmount method of lifecycle
        }
    }
    ReactDOM.render(<Clock/>, document.getElementById("root"))
  </script>
</body>
```

References

- ReactJS | State in React - GeeksforGeeks
- React Component Mounting And Unmounting - Learn.co
- Component Lifecycle | Build with React JS
- Rule | DeepScan