COURSEWARE

Lifecycle

Contents

Overview

In this module, we will discuss the different React Component Lifecycle methods that exist.

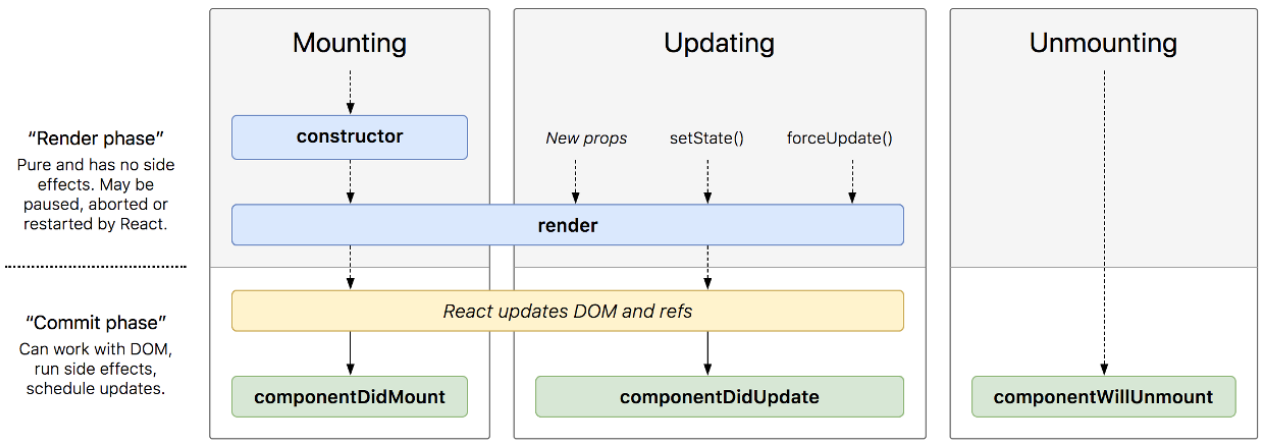
Component Lifecycle

The lifecycle of a component begins when it is first rendered in the DOM. It exists as long as it is represented in the DOM, until it is removed, and each component has its own lifecycle.

Aside from render, there are three types of lifecycle methods:

1. Mounting - Called when an instance of a component is created and inserted into the DOM.
2. Updating - Called when a component is being re-rendered, usually because of a change to props or state.
3. Unmounting - Called when a component is being removed from the dom.

Below is a simple lifecycle for class components



Methods are prefixed with **will** or **did**, indicating if they are executed **before** the event happens, or **after** the event happened.

There is also an error handling lifecycle method, which isn't really a lifecycle method but it occurs when an error occurs during a render.

Render

`Render()` is required in all components.
It should examine props and state, and return a single element; however, this element can have other elements or components nested inside it.
It can return null or false, indicating there is nothing to render.
It should not change components state. Render tends to return the same result every time it is called and does not interact with the browser.

```
render(){
    return(
        <Hello/>
    );
}
```

Mounting Methods

Mounting methods are called when an instance of a component is being created and inserted into the DOM.

Constructor()

The `constructor` method is... well, a constructor for a react component.
It should have a call to `super()` before any other code.
The correct place to initialise `state` is within the constructor.

```
constructor(){
    super();
    this.state = {
        header: "Default Header",
        content: "Default Content"
    };
}
```

componentWillMont()

`componentWillMont()` method will be invoked immediately before mounting occurs.

It is called before `render()` so changes in **state** do not trigger re-rendering.

Generally, it is recommended to use a constructor instead, however you may still see this method as it is still exists for backward compatibility.

```
componentWillMount(){
    // Advised to use constructor instead.
}
```

componentDidMont()

`componentDidMont()` is called immediately after a component is mounted.

It is ideal to implement making requests to the backend, within this method.

Calling a `setState()` method here will cause the component to re-render.

```
componentDidMount(){
    // Perfect place to do Axios / API calls
}
```

Mounting methods - State Call Order:

```
class App extends Component{
    constructor(){
        // First (including retrieving props and setting state)
    }
    componentWillMount(){
        // Second
    }
    render(){
        return(
            <div> // Third </div>
        );
    }
    componentDidMount(){
        // Fourth
    }
}
```

Updating Methods

Updating methods are called when a component is being re-rendered after changes to props or state.

componentWillReceiveProps(nextProps)

`componentWillReceiveProps(nextProps)` method is called before a mounted component receives new props.

The parameter that we pass into the functions are the new props.

It is useful for comparing new props with old to decide if the state should be changed

```
componentWillReceiveProps(nextProps){
    if(nextProps.dogName !== this.props.dogName){
        // Update the state with new dog name
    }
}
```

shouldComponentUpdate(nextProps, nextState)

This method is used to keep code efficient.

The default behaviour is to re-render on every state change

It is invoked before rendering when new props or state are being received, defaulting to true.
Returned false prevents `componentWillUpdate()`, `render()`, and `componentDidUpdate()` from being called.
Returning false doesn't prevent child components from re-rendering when their state changes.

```
shouldComponentUpdate(nextProps, nextState){
    return this.nextProps.dogName != this.props.dogName && nextState.input !==
this.state.input;
}
```

componentWillUpdate(nextProps, nextState)

This method is invoked immediately before rendering when new props or state are being received. It is an opportunity to preform preparation before an update occurs.

As it is an update method, it is not called during the component's initial render. We cannot use `this.state()` in this method.

Updating state in response to a prop change should be done in the `componentWillReceiveProps()` method instead. Similarly to `componentWillMount()`, it is advised to not use this method.

```
componentWillUpdate(nextProps, nextState) {
// don't use this one, similar concept to componentWillMount()
}
```

componentDidUpdate(nextProps, nextState)

This method is invoked immediately after an update occurs. It is an opportunity to operate on the DOM after a component update.

It is not called during the component's initial render, it's not invoked if `shouldComponentUpdate()` return false.

Updating Methods - State Call Order:

```
shouldComponentUpdate() {
    //first
}
componentWillUpdate() {
    //second
}
render() {
    return (
        <div>
            //Third
        </div>
    );
    }
componentDidUpdate() {
    //Fourth
}
```

Updating Methods - Props Call Order:

```
componentWillReceiveProps() {
    //first
}
shouldComponentUpdate() {
    //second
}
componentWillUpdate() {
    //third
}
render() {
    return (
        <div>
            //fourth
        </div>
    );
}
componentDidUpdate(){
    //fifth
}
```

Unmounting Methods

Unmounting methods are called when a component is being removed from the DOM.

componentWillUnmount()

`componentWillUnmount()` is invoked immediately before a component is unmounted and destroyed.
There is the opportunity to perform necessary clean up, e.g. invalidating timers, cleaning up DOM elements created in `componentDidMount()`

```
componentWillUnmount() {
    window.removeEventListener("resize",this.resizeListener);
}
```

Tutorial

In this tutorial, we will be putting most of the lifecycle methods into practice.

1. Create a class called `Clock` that extends `Component`:

```
import {Component} from 'react';
export default class Clock extends Component{

}
```

2. Create a constructor with `props` passed as parameter - call `super(props)` straight after:

```
constructor(props){
    super(props)
}
```

3. Set state of the following properties:

    - date: new Date()
    - text: ''
    - boolForShould: true

```
constructor(props){
   super(props);
   this.state={
        date: new Date(),
        text: "",
        boolForShould: true
   }
}
```

4. Place a `console.log()` statement in the constructor printing the state value of `date`:

```
console.log("Constructor sets the time as : "+ this.state.date);
```

5. Create a `componentWillMount()` method - print the state value of date within it:

```
componentWillMount(){
    console.log(`ComponentWillMount sets the time as` + this.state.date);
}
```

6. Create a `componentDidMount()` method - print the state value of date within it; create a variable called timerID which is initialised to the `setIntervalMethod` that calls a function called tick after 1 second:

```
componentDidMount() {
    console.log("componentDidMount sets the time as :" + this.state.date);
    this.timerID = setInterval(() => this.tick(), 1000);
}
```

7. Create a `componentWillUpdate()` method that prints the value of the state date:

```
componentWillUpdate() {
    console.log("componentWillUpdate sets the time as :" +
this.state.date);
}
```

8. Create a `componentWillUnmount()` method which prints the state of date and calls `clearInterval()` on the timerId:

```
 componentWillUnmount() {
     console.log("componentWillUnmount sets the time as :" +
this.state.date);
     clearInterval(this.timerID);
 }
```

9. Create a `shouldComponentUpdate()` method that prints the state of date, and also returns the value of boolForShould:

```
 shouldComponentUpdate() {
     console.log("shouldComponentUpdate sets the time as :" +
this.state.date);
     return this.state.boolForShould;
 }
```

10. Create a `componentDidUpdate()` method which prints the state of date:

```
componentDidUpdate() {
    console.log("componentDidUpdate sets the time as :" + this.state.date);
}
```

11. Create a method called `tick()` which sets the state of date to a new Date object:

```
tick(){
    this.setState({ date : new Date() });
}
```

12. Add the following three methods:

```
stateHandle = () => {
    console.log("stateHandle sets the time as :" + this.state.date);
    this.setState({
        text: "Updated"
    });
};

stateHandle2 = () => {
    console.log("stateHandle2 sets the time as :" + this.state.date);
    clearInterval(this.timerID);
    this.setState({
        text: "Updated and timer has stopped"
    });
};

shouldHandle = () => {
    console.log("shouldHandle sets the time as :" + this.state.date);
    this.setState({
        boolForShould: !this.state.boolForShould
    });
};
```

13. Create a render method which returns:

    1. `<h1>` - With the state value of `text`
    2. `<h2>` - With the state value of `date.toLocaleTimeString()`
    3. `<button>` - onClick calls `stateHandle`
    4. `<button>` - onClick calls `stateHandle2`
    5. `<button>` - onClick calls `shouldHandle`

```
render() {
    return (
        <div>
            <h1>{this.state.text}</h1>
            <h2>It is {this.state.date.toLocaleTimeString()}</h2>
            <button onClick={this.stateHandle}>Changes State</button>
            <button onClick={this.stateHandle2}> Changes State and stops
timer </button>
            <button onClick={this.shouldHandle}> Changes
shouldComponentUpdate </button>
        </div>
    );
}
```

▶ Clock.jsx

Exercises

There are no exercises for this module.