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).

Example:

The constructor method is called, by React, every time you make a component:

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

render() {

return (

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

);

}

}

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

Output:

**My Favorite Color is red**

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:

Example:

The getDerivedStateFromProps method is called right before the render method:

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>

);

}

}

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

Output:

**My Favorite Color is yellow**

render

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

Example:

A simple component with a simple render() method:

class Header extends React.Component {

render() {

return (

<h1>This is the content of the Header component</h1>

);

}

}

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

Output:

**This is the content of the Header component**

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.

Example:

At first my favorite color is red, but give me a second, and it is yellow instead:

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>

);

}

}

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

Output:  
  
In Beginning:

**My Favorite Color is red**

After 1 sec:

**My Favorite Color is yellow**

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:

Example:

If the component gets updated, the getDerivedStateFromProps() method is called:

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>

);

}

}

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

/\*

This example has a button that changes the favorite color to blue,

but since the getDerivedStateFromProps() method is called,

the favorite color is still rendered as yellow

(because the method updates the state

with the color from the favcol attribute).

\*/

**My Favorite Color is yellow**

Change color

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:

Example:

Stop the component from rendering at any update:

class Header 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>

);

}

}

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

**Output:**

**My Favorite Color is red**

Change color

Example:

Same example as above, but this time the shouldComponentUpdate() method returns true instead:

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

shouldComponentUpdate() {

return true;

}

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>

);

}

}

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

Output:

**My Favorite Color is red**

Change color

After clicking:  
**My Favorite Color is blue**

Change color

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:

Example:

Click the button to make a change in the component's state:

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>

);

}

}

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

Output:

**My Favorite Color is red**

Change color

After clicking:  
**My Favorite Color is blue**

Change color

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:

Example:

Use the getSnapshotBeforeUpdate() method to find out what the state object looked like before the update:

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

componentDidMount() {

setTimeout(() => {

this.setState({favoritecolor: "yellow"})

}, 1000)

}

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>

);

}

}

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

Output:

**My Favorite Color is yellow**

Before the update, the favorite was red

The updated favorite is yellow

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:

Example:

The componentDidUpdate method is called after the update has been rendered in the DOM:

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

componentDidMount() {

setTimeout(() => {

this.setState({favoritecolor: "yellow"})

}, 1000)

}

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>

);

}

}

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

Output:  
At beginning:

**My Favorite Color is red**

After 1 sec:  
**My Favorite Color is yellow**

The updated favorite is yellow

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.

Example:

Click the button to delete the header:

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>

);

}

}

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

Output:  
**Hello World!**

Delete Header

After clicking button:  
Delete Header