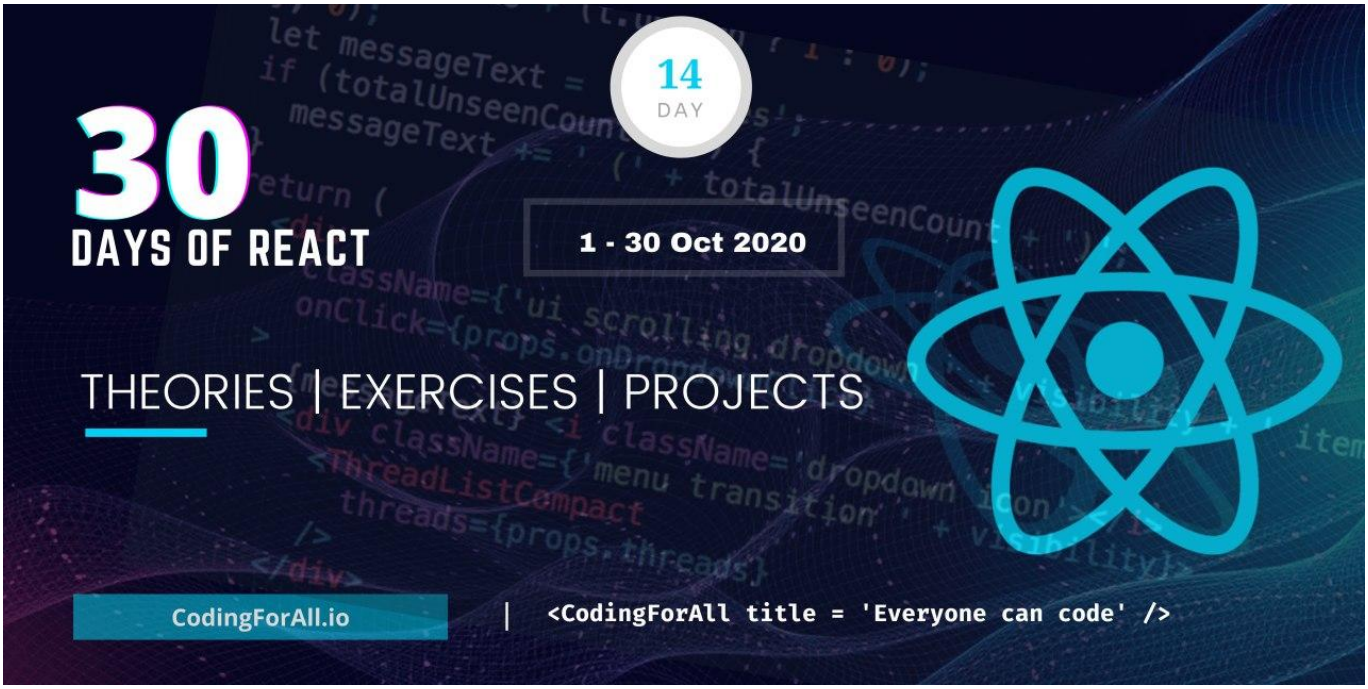
30 Days Of React:Component Life Cycles

Author: Asabeneh Yetayeh

October, 2020

<< Day 13 | Day 15 >>



Component Life Cycles

What is component life cycle

Component life cycle is the process of mounting, updating and destroying a component in a React application. You can associate a component life cycle with the process of human growth:birth, adult, elderly and death. In React component also a component can be mounted or rendered the first time, can be updated by changing the data and also can be destroyed whenever it is not needed. In React each component has three main phases:

- Mounting
- Updating
- Unmounting

Mounting

Rendering or putting React component into the DOM is called mounting. The following built-in methods run in the given order during mounting of a React component.

1. constructor()
2. static getDerivedStateFromProps()
3. render()
4. componentDidMount()

When we have been making a class-based component we used a built-in render method and it is required in all class-based components but other methods are optional. See the order of execution of the different methods by running the following snippet of codes.

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

class App extends Component {
  constructor(props) {
    super(props)
    console.log('I am  the constructor and  I will be the first to run.')
    this.state = {
      firstName: '',
    }
  }

  static getDerivedStateFromProps(props, state) {
    console.log(
      'I am getDerivedStateFromProps and I will be the second to run.'
    )
    return null
  }
  componentDidMount() {
    console.log('I am componentDidMount and I will be last to run.')
  }

  render() {
    console.log('I am render and I will be the third to run.')
    return (
```

```
        <div className='App'>
          <h1>React Component Life Cycle</h1>
        </div>
      )
    }
  }

  const rootElement = document.getElementById('root')
  ReactDOM.render(<App />, rootElement)
```

Contructor

Nowadays we write class based-component without a constructor and we can write the state also outside the constructor. In older version React we the state used be always inside the constructor.

The constructor() method is executed before any other methods, when component is initiated and it is the place where to set the initial state and other values. In class we use constructor parameter to inherit from parents and in React to the constructor take a props parameter and the super method has to be also called. Look at the snippet of code about constructor and state.

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

class App extends Component {
  constructor(props) {
    super(props)
    console.log('I am  the constructor and  I will be the first to run.')
    this.state = {
      firstName: '',
    }
  }
  render() {
    return (
      <div className='App'>
        <h1>React Component Life Cycle</h1>
        <h2>The constructor is the first to Run</h2>
        <p>Author:{this.state.firstName}</p>
      </div>
    )
  }
}

const rootElement = document.getElementById('root')
ReactDOM.render(<App />, rootElement)
```

getDerivedStateFromPros

As we can understand from the name, this method derives a state from props. The getDerivedStateFromProps() method is called right before rendering the component in the DOM. This the

right place to set the state object based on the initial props.

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

const User = ({ firstName }) => (
  <div>
    <h1>{firstName}</h1>
  </div>
)

class App extends Component {
  constructor(props) {
    super(props)
    console.log('I am  the constructor and  I will be the first to run.')
    // we can write state inside or outside the constructor
    // if is written outside the constructor it does not need the keyword this
    this.state = {
      firstName: 'John',
    }
  }
  static getDerivedStateFromProps(props, state) {
    console.log(
      'I am getDerivedStateFromProps and I will be the second to run.'
    )
    return { firstName: props.firstName }
  }

  render() {
    return (
      <div className='App'>
        <h1>React Component Life Cycle</h1>
        <h3>getDerivedStateFromProps</h3>
        <User firstName={this.state.firstName} />
      </div>
    )
  }
}

const rootElement = document.getElementById('root')
ReactDOM.render(<App firstName='Asabeneh' />, rootElement)
```

Render

The render method is a required method when we create a class-based component. The render method is where we return JSX. The render methods render whenever there is change in state. Do not set your state inside render method.

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

```
const User = ({ firstName }) => (
  <div>
    <h1>{firstName}</h1>
  </div>
)

class App extends Component {
  constructor(props) {
    super(props)
    console.log('I am  the constructor and  I will be the first to run.')
    // we can write state inside or outside the constructor
    // if is written outside the constructor it does not need the keyword this
    this.state = {
      firstName: 'John',
    }
  }
  render() {
    // Never do this
    // Do not reset inside the render method, create a method to reset the state

    return (
      <div className='App'>
        <h1>React Component Life Cycle</h1>
        <h3>Render method</h3>
      </div>
    )
  }
}

const rootElement = document.getElementById('root')
ReactDOM.render(<App firstName='Asabeneh' />, rootElement)
```

ComponentDidMount

As we can understand the name of the method that this method called after component is render. This a place place to setting time interval and calling API. Look at the following setTimeout implementation in componentDidMount method.

```
import React, { Component } from 'react'
import ReactDOM from 'react-dom'
class App extends Component {
  constructor(props) {
    super(props)
    console.log('I am  the constructor and  I will be the first to run.')
    this.state = {
      firstName: 'John',
    }
  }
  componentDidMount() {
    console.log('I am componentDidMount and I will be last to run.')
```

```
    // after 3 seconds it resets the state
    setTimeout(() => {
      this.setState({
        firstName: 'Asabeneh',
      })
    }, 3000)
  }

  render() {
    return (
      <div className='App'>
        <h1>React Component Life Cycle</h1>
        <h2>componentDidMount Method</h2>
        {this.state.firstName}
      </div>
    )
  }
}

const rootElement = document.getElementById('root')
ReactDOM.render(<App />, rootElement)
```

In the above snippet of code, we saw how to implement setTimeout inside a componentDidMount method. In next example, we will implement an API call using fetch.

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

class App extends Component {
  constructor(props) {
    super(props)
    console.log('I am  the constructor and  I will be the first to run.')
    this.state = {
      firstName: 'John',
      data: [],
    }
  }

  componentDidMount() {
    console.log('I am componentDidMount and I will be last to run.')
    const API_URL = 'https://restcountries.eu/rest/v2/all'
    fetch(API_URL)
      .then((response) => {
        return response.json()
      })
      .then((data) => {
        console.log(data)
        this.setState({
          data,
        })
      })
```

```
      .catch((error) => {
        console.log(error)
      })
  }

  render() {
    return (
      <div className='App'>
        <h1>React Component Life Cycle</h1>
        <h1>Calling API</h1>
        <div>
          <p>There are {this.state.data.length} countries in the api</p>
          <div className='countries-wrapper'>
            {this.state.data.map((country) => (
              <div>
                <div>
                  {' '}
                  <img src={country.flag} alt={country.name} />{' '}
                </div>
                <div>
                  <h1>{country.name}</h1>
                  <p>Capital: {country.capital}</p>
                  <p>Population: {country.population}</p>
                </div>
              </div>
            ))}
          </div>
        </div>
      </div>
    )
  }
}

const rootElement = document.getElementById('root')
ReactDOM.render(<App />, rootElement)
```

Sometimes it is better to have a separate method to render the data. See the example below:

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

class App extends Component {
  constructor(props) {
    super(props)
    console.log('I am  the constructor and  I will be the first to run.')
    this.state = {
      firstName: 'John',
      data: [],
    }
  }

  componentDidMount() {
```

```
    console.log('I am componentDidMount and I will be last to run.')
    const API_URL = 'https://restcountries.eu/rest/v2/all'
    fetch(API_URL)
      .then((response) => {
        return response.json()
      })
      .then((data) => {
        console.log(data)
        this.setState({
          data,
        })
      })
      .catch((error) => {
        console.log(error)
      })
  }
  renderCountries = () => {
    return this.state.data.map((country) => {
      return (
        <div>
          <div>
            {' '}
            <img src={country.flag} alt={country.name} />{' '}
          </div>
          <div>
            <h1>{country.name}</h1>
            <p>Population: {country.population}</p>
          </div>
        </div>
      )
    })
  }

  render() {
    return (
      <div className='App'>
        <h1>React Component Life Cycle</h1>
        <h1>Calling API</h1>
        <div>
          <p>There are {this.state.data.length} countries in the api</p>
          <div className='countries-wrapper'>{this.renderCountries()}</div>
        </div>
      </div>
    )
  }
}

const rootElement = document.getElementById('root')
ReactDOM.render(<App />, rootElement)
```

Updating

After a component has been mounted on the DOM, it can be updated when a state or props change. An update of a React component can be caused by changes to props or state . These methods are called in the following order when a component is being re-rendered:

1. static getDerivedStateFromProps()
2. shouldComponentUpdate()
3. render()
4. getSnapshotBeforeUpdate()
5. componentDidUpdate()

getDerivedStateFromProps

Similar to the mounting phase, getDerivedStateFromProps can be also called in the updating phase. The getDerivedStateFromProps is the first method that is called when a component gets updated.

shouldComponentUpdate

The shouldComponentUpdate() built-in life cycle method should return a boolean. If this method does not return true the application will not update.

If the method does not return true the application will never update. This can be used for instance to block use when it reaches to a certain point(game, subscription) or may be to block a certain user.

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

class App extends Component {
  constructor(props) {
    super(props)
    console.log('I am  the constructor and  I will be the first to run.')
    this.state = {
      firstName: 'John',
      data: [],
    }
  }

  shouldComponentUpdate(nexProps, nextState) {
    console.log(nextProps, nextState)
    // if the return is true, the application will never update.
    return true
  }

  render() {
    return (
      <div className='App'>
        <h1>React Component Life Cycle</h1>
      </div>
    )
  }
}
```

```
const rootElement = document.getElementById('root')
ReactDOM.render(<App />, rootElement)
```

For instance, if we want to stop doing challenge after 30 days we can increment the day from 1 to 30 and we can block the application at day 30. Look the example.

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

class App extends Component {
  constructor(props) {
    super(props)
    console.log('I am  the constructor and  I will be the first to run.')
    this.state = {
      firstName: 'John',
      day: 1,
    }
  }

  shouldComponentUpdate(nextProps, nextState) {
    console.log(nextProps, nextState)
    console.log(nextState.day)
    if (nextState.day > 31) {
      return false
    } else {
      return true
    }
  }
  // the doChallenge increment the day by one
  doChallenge = () => {
    this.setState({
      day: this.state.day + 1,
    })
  }
  render() {
    return (
      <div className='App'>
        <h1>React Component Life Cycle</h1>
        <button onClick={this.doChallenge}>Do Challenge</button>
        <p>Challenge: Day {this.state.day}</p>
        {this.state.congratulate && <h2>{this.state.congratulate}</h2>}
      </div>
    )
  }
}

const rootElement = document.getElementById('root')
ReactDOM.render(<App />, rootElement)
```

render

As we have mentioned it on the mounting phase of the component, the render() method is called when a component gets updated. It has to re-render the HTML to the DOM, with the new changes.

componentDidUpdate

The componentDidUpdate method takes two parameters: the prevProps and prevState. It is called after the component is updated in the DOM.

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

class App extends Component {
  constructor(props) {
    super(props)
    console.log('I am  the constructor and  I will be the first to run.')
    this.state = {
      firstName: 'John',
      data: [],
    }
  }
  componentDidUpdate(prevProps, prevState) {
    console.log(prevState, prevProps)
  }
  render() {
    return (
      <div className='App'>
        <h1>React Component Life Cycle</h1>
      </div>
    )
  }
}

const rootElement = document.getElementById('root')
ReactDOM.render(<App />, rootElement)
```

Let's use the above two life cycle methods together.

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

class App extends Component {
  constructor(props) {
    super(props)
    console.log('I am  the constructor and  I will be the first to run.')
    this.state = {
      day: 1,
      congratulate: '',
    }
  }
```

```
  shouldComponentUpdate(nextProps, nextState) {
    console.log(nextProps, nextState)
    console.log(nextState.day)
    if (nextState.day > 31) {
      return false
    } else {
      return true
    }
  }

  doChallenge = () => {
    this.setState({
      day: this.state.day + 1,
    })
  }
  componentDidUpdate(prevProps, prevState) {
    if (prevState.day == 30) {
      this.setState({
        congratulate: 'Congratulations,Challenge has been completed',
      })
    }
    console.log(prevState, prevProps)
  }

  render() {
    return (
      <div className='App'>
        <h1>React Component Life Cycle</h1>
        <h1>Calling API</h1>
        <button onClick={this.doChallenge}>Do Challenge</button>
        <p>Challenge: Day {this.state.day}</p>
        {this.state.congratulate && <h2>{this.state.congratulate}</h2>}
      </div>
    )
  }
}

const rootElement = document.getElementById('root')
ReactDOM.render(<App />, rootElement)
```

Unmounting

The final phase in the lifecycle of a component is unmounting. The unmounting phase removes component from the DOM. The componentWillUnmount method is the only built-in method that gets called when a component is unmounted.

Exercises

Exercises: Level 1

1. What is component life cycles
2. What is the purpose of life cycles
3. What are the three stages of a component life cycle
4. What does mounting means?
5. What does updating means
6. What does unmounting means?
7. What is the most common built-in mounting life cycle method?
8. What are the mounting life cycle methods?
9. What are the updating life cycle methods?
10. What is the unmounting life cycle method?

Exercises: Level 2

Coming

Exercises: Level 3

Coming

🎉 CONGRATULATIONS ! 🎉

<< Day 13 | Day 15 >>