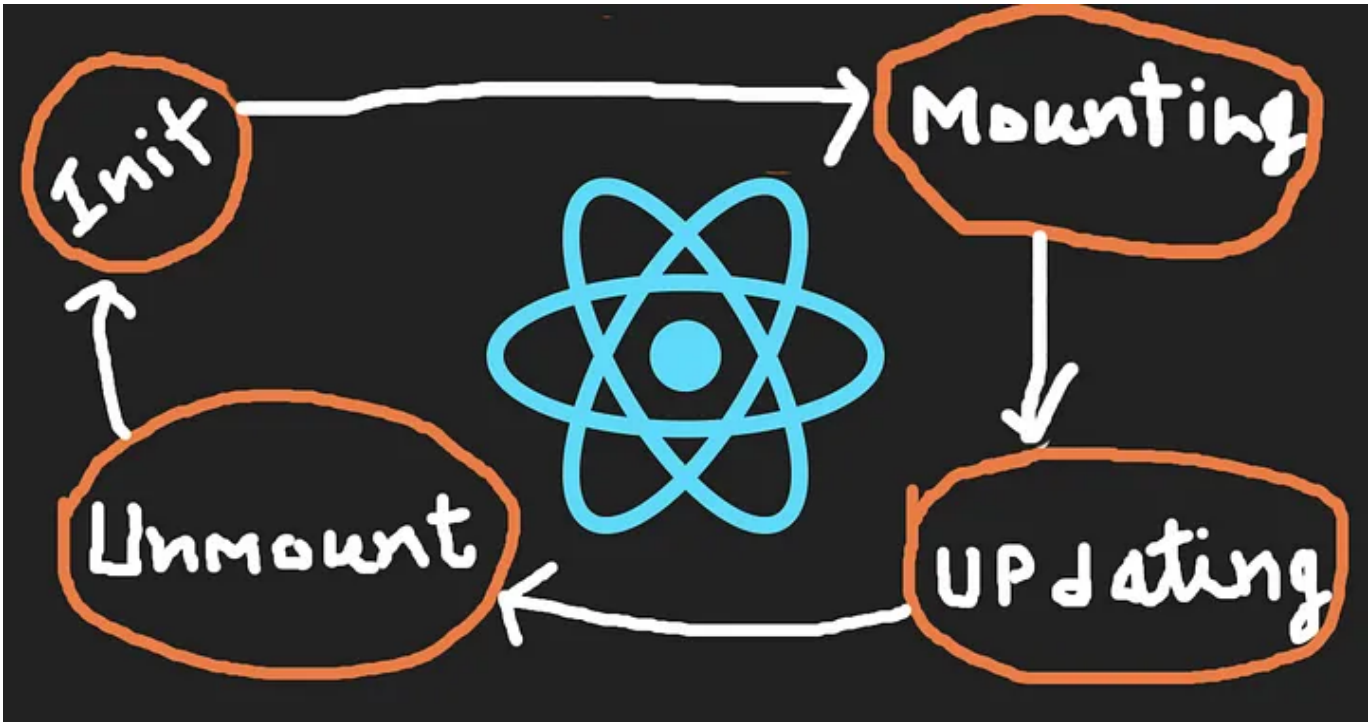
Sanjeev Shukla Follow
Nov 4, 2019 · 4 min read · Listen

Understanding React — Component life-cycle Methods(React 16.3+)



ReactJs v16.3 introduced a lot of changes in component lifecycle methods. There have been a few deprecations of methods as well as some addition of new methods. Let's discuss it in 3 fundamental questions What, Why and How?

What is new?

The release of 16.3 introduced some new life-cycle functions, which replace existing ones to provide better support for the new asynchronous nature of React. there are two new methods in v16.3.
1- getDerivedStateFromProps()
2- getSnapshotBeforeUpdate()

What has removed?

1- componentWillMount()
2- componentWillReceiveProps()
3- componentWillUpdate()
However, they are not fully going away as you'll be able to use them with as UNSAFE_componentWillMount, UNSAFE_componentWillRecieveProps, UNSAFE_componentWillUpdate from version 17.

Why those methods removed?

usually, components are composed of UI +Data, and because of data is always dynamic nature, you may have to show the different UIs on different times. To handle this react gives life cycle methods These methods give us opportunities to control and manipulate how a component reacts to changes in the application. it allows us to update the UI and application or control the behavior of a component over time.
The original lifecycle model was not designed for some of the upcoming async features like the asynchronous rendering of the component. With the introduction of new rendering features, some of these lifecycle methods can be unsafe if used.
For example, asynchronous rendering may cause componentWillMount to trigger multiple times of your component tree and that can directly impact performance.

OK! How to use new life cycle methods?

Simple! There are three phases of a component as —
Mounting ⇒ Updating ⇒ Unmounting.

1- Mounting

The mounting means injecting the elements into the DOM.
React has four built-in methods that get called, in this order, when mounting a component:
1- constructor()
2- getDerivedStateFromProps()
3- render()
4- componentDidMount()

Constructor —

The constructor() method is called before any other method. when the component is initiated, the contractor is the natural place to set up the initial state and other default values.

```
class App extends Component {
  constructor(props) {
    super(props);
    this.state = {value: "blue"};
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.value}</h1>
    );
  }
}
ReactDOM.render(<App />, document.getElementById('root'));
```

constructor.js hosted with ❤ by GitHub view raw

Initialization in constructor

getDerivedStateFromProps —

The getDerivedStateFromProps() method is called just before rendering the element on the DOM. It takes two arguments, props and state then return an object with changes to the state.
This function should return an object to update state or null because the static function will not have any acknowledge to this object. This is the right place to sync up your props and state.
The example below starts with the favorite fruit being “apple”, but the getDerivedStateFromProps() method updates the favorite fruit based on the newChoice prop:

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {myFruit: "Apple"};
  }
  static getDerivedStateFromProps(props, state) {
    return {myFruit: props.newChoice };
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.myFruit}</h1>
    );
  }
}
ReactDOM.render(<App newChoice="orange"/>, document.getElementById('root'));
```

getDerivedStateFromProps.js hosted with ❤ by GitHub view raw

render

he render() method is required and is the method that actual outputs HTML to the DOM.

```
class App extends React.Component {
  constructor() {
  // 1- first it will called
  }
  getDerivedStateFromProps() {
    // 2- it will be called after constructor
  }
  render() {
    // 3- it will be called after getDerivedStateFromProps method
    return (
      <h1>Some Text</h1>
    );
  }
}

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

Render_Initialization.js hosted with ❤ by GitHub view raw

componentDidMount

The componentDidMount() method is called just after the component is rendered. This is the place where you run statements that require that the element is already placed in the DOM. for example event bindings.

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {animal: "dog"};
  }
  componentDidMount() {

  // after mounting the component it will change the state after 1 sec.
    setTimeout(() => {
      this.setState({animal: "cat"})
    }, 1000)
  }
  render() {
    return (
      <h1>My Favorite Animal is {this.state.animal}</h1>
    );
  }
}
ReactDOM.render(<App />, document.getElementById('root'));
```

componentDidMount_Initialization.js hosted with ❤️ by GitHub view raw

2- Updating

The next phase in the lifecycle is when a component is updated. A component jumps into updating state whenever there is a change in the component's state or props. React has five built-in methods that get called, in this order, when a component is updated.

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

getDerivedStateFromProps

getDerivedStateFromProps() method is called on both events, mounting and updating . It is also called if the parent component is re-render. In summary, this function will be called when —

- The component is initialized.
- Receiving new props whether they are changed or not.
- The parent component is re-rendered

This is the natural place to set the state object based on the updated props.
The example below has a button that changes the favorite place , but since the getDerivedStateFromProps() method is called, which mutate the state with the place from the props.

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritePlace: "NY"};
  }
  static getDerivedStateFromProps(props, state) {
    return {favoritePlace: props.favPlace };
  }
  changePlace = () => {
    this.setState({favoritePlace: "Sidny"});
  }
  render() {
  // always return mumbai no react on button event
    return (
      <div>
      <h1>My Favorite Place is {this.state.favoriteplace}</h1>
      <button type="button" onClick={this.changePlace}>Change Place</button>
      </div>
    );
  }
}

ReactDOM.render(<App favPlace="mumbai"/>, document.getElementById('root'));
```

getDerivedStateFromProps_Updating.js hosted with ❤ by GitHub view raw

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:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritActor: "Leonardo"};
  }
  shouldComponentUpdate() {
    return false;
  }
  changeActor = () => {
    this.setState({favoriteActor: "Tom Cruise"});
  }


render() {
    // this will never update as shouldUpdate method is returning false, it will always showing Leonardo
    return (
      <div>
      <h1>My Favorite Actor is {this.state.favoriteActor}</h1>
      <button type="button" onClick={this.changeActor}>Change Actor</button>
      </div>
    );
  }
}
ReactDom.render(<App />, document.getElementbyId('root'));
```

shouldComponentUpdate_updating.js hosted with ❤️ by GitHub view raw

render

he 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.

getSnapshotBeforeUpdate

the update is triggered now if you want to access earlier props then you can use this method, 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.

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoriteMovie: "Avengers"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoriteMovie: "Spiderman"})
    }, 1000)
  }
  getSnapshotBeforeUpdate(prevProps, prevState) {
    console.log( prevState.favoriteMovie); // here you can access the previous value of favoriteMovie
  }
  componentDidUpdate() {
    console.log(this.state.favoriteMovie); //it will print the updated value
  }
  render() {
    return (
      <div>
        <h1>Learn Life cycle methods</h1>
      </div>
    );
  }
}

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

getSnapshotBeforeUpdate_updating.js hosted with ❤️ by GitHub view raw

componentDidUpdate

The componentDidUpdate method is called after the component is updated in the DOM.

```jsx
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoriteCountry: "India"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoriteCountry: "China"})
    }, 1000)
  }
  componentDidUpdate() {
    console.log(this.state.favoriteCountry);
  }
  render() {
    return (
      <div>
        <h1>My Favorite Country is {this.state.favoriteCountry}</h1>
      </div>
    );
  }
}

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

componentDidUpdate_updating.js hosted with ❤️ by GitHub view raw

Unmounting

The next phase in the lifecycle is when a component is removed from the DOM. React has only one built-in method that gets called when a component is unmounted.

1- componentWillUnmount()

When a component is unmounted or destroyed, this method will be called. This is a place to do some clean up like —

- Invalidating timers
- Canceling any network request
- free the connections if using
- Remove event handlers
- Clean up any event subscriptions
- Calling setState here is useless, there will be no re-rendering.

```
export default class ClockExample extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      date: new Date()
    };
  }
  tick() {
    this.setState({
      date: new Date()
    });
  }
  componentDidMount() {
    this.timerID = setInterval(() => {
      this.tick();
    }, 1000);
  }
  componentWillUnmount() {
    clearInterval(this.timerID);
  }
  render() {
    return (
        <div>It is {this.state.date.toLocaleTimeString()}.</div>
    );
  }
}
```

componentWillUnmount_unmounting.js hosted with ❤️ by GitHub view raw

there is one more life cycle method in react to handle the errors —

componentDidCatch

Since React 16, a new life-cycle method has been introduced in order to catch and handle uncaught errors happening in the child components. This will enable us to render the fallback UI when needed instead of crashing the component.

```
componentDidCatch(error, info) {
   this.setState({ hasError: true });
 }
 render() {
   if (this.state.hasError) {
    //custom fallback UI
     return <div>Something went wrong.</div>;
   }
   return this.props.children; //render children
 }
```

componentDidCatch_error.js hosted with ❤ by GitHub view raw

> *Hope this article woud be helpful to understand react life cycles. Happy Learning !*

React React Lifecycle Method React Lifecycle Getderivedstatefromprops React New Feature