Component Lifecycle Methods

Introduction

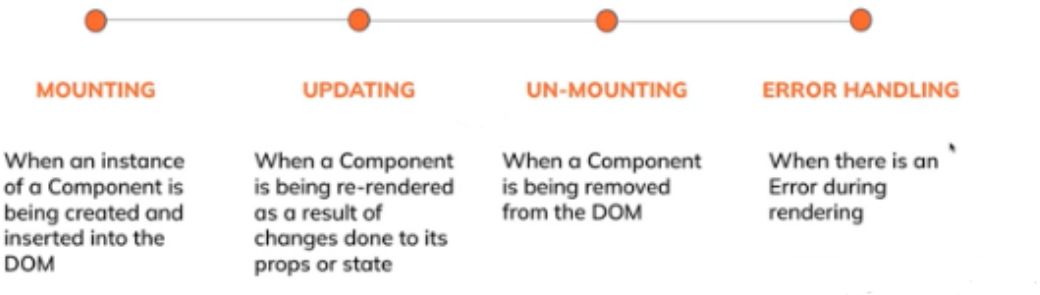
Lifecycle

Lifecycle is the series of stages through which a component passes from the beginning of its life until its death. The life of the React component starts when it is born (Created/Mounted) and ends when it is destroyed (Unmounted).

Different Phases of a Lifecycle

Different Phases of a component lifecycle are:

- **Mounting:** When a component is being created and inserted into the DOM.
- **Updating:** When a Component is being re-rendered due to any updates made to its state or props.
- **Unmounting:** When it is destroyed/ removed from the DOM.
- **Error Handling:** When there is an error during rendering.



During the lifecycle of a component, certain methods are called at each phase where we can execute some logic or perform a side-effect.

Side effects are actions that are not predictable because they are actions that are performed with the "outside world."

For example: Using Browser APIs like localStorage, using the native DOM methods instead of the ReactDOM, fetching the data from an API, and setting timeouts and intervals.

Mounting Phase

These methods are called in the following sequence when an instance of a component is being created:

- `constructor()`
- `static getDerivedStateFromProps()`
- `render()`
- `componentDidMount()`

constructor

- A special function that will get called whenever a new component is created.
- It can be used to initialize the state and bind the event handlers.
- This is the only place where the state can be modified directly. Everywhere else state should be updated using the `setState` function (used to update the state of a component).
- Avoid introducing any side effects/subscriptions in the constructor.

Example:

```
class Counter extends Component {
    constructor(props) {
        super(props);
        this.state = {
            name: "Counter",
            count: 0
         }
        console.log("Counter Constructor")
    }
}
```

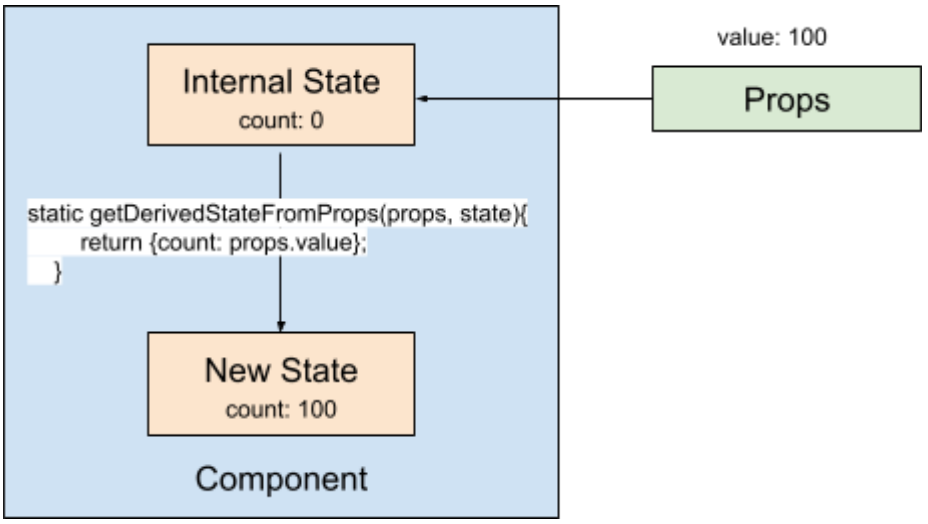
static getDerivedStateFromProps

- It is invoked right before the render function. This method is invoked in both the mounting and updating phases.

- This method exists for rare use cases where the state depends on changes in the props over time.If there is no change in state, then this method returns the null value.
- It is a static method that does not have any access to this keyword.

Example:

It will return an object to update the state. if the props.value is equal to state.counter then it will return a null value.




```
static getDerivedStateFromProps(props, state){
    console.log("Counter getDerivedStateFromProps");
    return {count: props.value};
}
```

render()

- This is the only required method in the class component. `render()` executes during both the mounting and updating phase of the component's lifecycle.
- It is used to render elements to the DOM by returning some JSX.
- The `render()` method must be a pure function, meaning it should not modify the component's state, as when the state gets updated, the render method gets automatically called, which could lead to infinite looping.
- `render()` will not be invoked if `shouldComponentUpdate()` returns false.

Example:

It returns JSX to render your UI and returns null value if there is nothing to render inside the component.

```
render() {
    console.log("Counter Render")
    return (
        <h1>{this.state.count}</h1>
     );
}
```

componentDidMount

- It is invoked after a component is mounted. (initially renders on the screen).
- This method is a good place to handle side effects like setting up subscriptions and loading data from a remote endpoint.
- You can also use the `setState` function in this method to update the state.

Example:

When the state gets updated inside this method, it causes another rendering just before the browser updates the UI.

```
componentDidMount() {
  console.log("Counter componentDidMount");
  setTimeout(() => {
    this.setState({count: 50});
  }, 1000);
}
```

Updating Phase

Changes to props or state can cause an update. These methods are called in the following order when a component is being re-rendered:

- `static getDerivedStateFromProps()`
- `shouldComponentUpdate()`
- `render()`
- `getSnapshotBeforeUpdate()`
- `componentDidUpdate()`

shouldComponentUpdate()

- `shouldComponentUpdate()` is called as soon as `static getDerivedStateFromProps()` the method is invoked.
- In the `shouldComponentUpdate()` method, you can return a Boolean value that controls whether the component gets rerendered upon a change in state/props. It defaults to true.
- This method only exists as a performance optimization. Do not rely on it to "prevent" a rendering, as this can lead to bugs.

Example:

It returns the boolean value true or false if you want to re-render or not.

```
shouldComponentUpdate(nextProps, nextState) {
  console.log("Counter shouldComponentUpdate");
  if(this.state.count == nextState.count) {
    return false;
  }
  return true;
}
```

getSnapshotBeforeUpdate()

- `getSnapshotBeforeUpdate()` is invoked just after the render() method.
- It stores the previous values of the state after the DOM is updated, meaning that even after the update, you can check what the values were before the update. Any value returned by this lifecycle method will be passed as a parameter to `componentDidUpdate()`.
- Most likely, you'll rarely reach for this lifecycle method. But it comes in handy when you need to grab information from the DOM (and potentially change it) just after an update is made, like a chat thread that needs to handle the scroll position.
- A snapshot value (or null) should be returned.

Example:

```
getSnapshotBeforeUpdate(prevProps, prevState) {
  console.log("Counter getSnapshotBeforeUpdate");
  return prevState.time || null;
}
```

componentDidUpdate()

- `componentDidUpdate()` is invoked immediately after updating occurs.
- It can operate on the DOM when the component has been updated.
- This is also a good place to do network requests as long as you compare the current props to previous props (e.g., a network request may not be necessary if the props have not changed).
- It is similar to `componentDidMount()` as you can use `setState()` or fetch API call but you have to mention a condition to check if the previous state or props has changed or not.

Example:

```
getSnapshotBeforeUpdate(prevProps, prevState) {
  console.log("Counter getSnapshotBeforeUpdate");
  return prevState.count || null;
}

componentDidUpdate(prevProps, prevState, snapshot) {
  console.log("Counter componentDidUpdate");
  if (snapshot !== null) {
    this.setState({ count: 20 });
  }
}
```

Unmounting Phase

This method is called when a component is being removed from the DOM:

- `componentWillUnmount()`

componentWillUnmount()

- It is invoked immediately before a component is unmounted and destroyed.
- Perform any necessary cleanup in this method, such as invalidating timers, canceling network requests, or cleaning up any subscriptions that were created in `componentDidMount()`.
- You should not call `setState()` in `componentWillUnmount()` because the component will never be re-rendered. Once a component instance is unmounted, it will never be mounted again.

Example:

```
componentWillUnmount() {
  if (this.count) {
    clearInterval(this.timer);
  }
}
```

Error Handling Phase

These methods are called when an error occurs during rendering, in a lifecycle method, or the constructor of any child component.

- `static getDerivedStateFromError()`
- `componentDidCatch()`

static getDerivedStateFromError()

- This lifecycle is invoked after a descendant component has thrown an error.
- It receives the error thrown as a parameter and should return a value to update the state.
- `getDerivedStateFromError()` is called during the “render” phase, so side effects are not permitted. For those use cases, use `componentDidCatch()` instead.

Example:

whenever an error is thrown in a descendant component, the error will be logged to the console, console.error(error), and an object is returned from the getDerivedStateFromError method. This will be used to update the state of the ErrorBoundary component i.e., with hasError: true.

```
  static getDerivedStateFromError(error){
      console.log("Error:", error);
      return{
          hasError: true,
          error: error
      }
  }
```

componentDidCatch()

- This lifecycle is invoked after a descendant component has thrown an error. It receives two parameters:
    - **error** - The error that was thrown.
    - **info** - An object with a componentStack key containing information about which component threw the error.
- `componentDidCatch()` is called during the “commit” phase, so side effects are permitted. It should be used for things like logging errors.

Example:

```
    componentDidCatch(error, info){
        console.log("Error:", error);
        console.log("Error Info: ", info);
    }
```

Error boundaries

Error boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed.

Example:

```
import { Component } from "react";
class ErrorBoundary extends Component {
    constructor() {
        super();
        this.state = {
            hasError: false,
            error: ""
        }
    }

    static getDerivedStateFromError(error){
        return{
            hasError: true,
            error: error
        }
    }

    componentDidCatch(error, info){
```

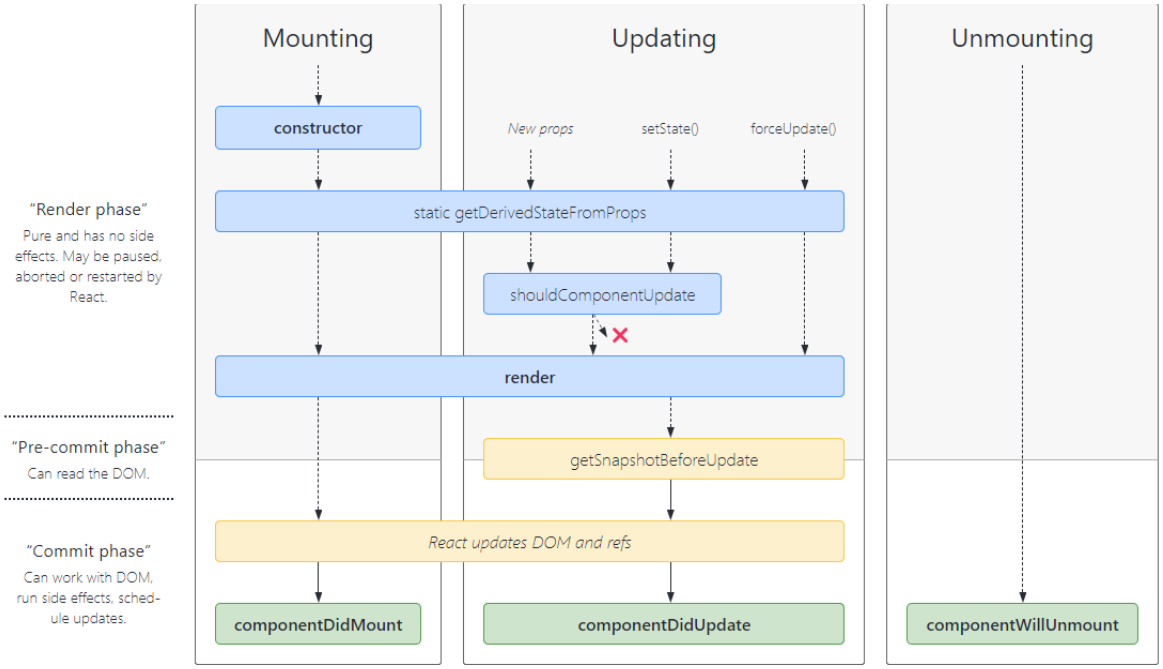
```
        console.log("Error:", error);
        console.log("Error Info: ", info);
    }

    render() {
        if(this.state.hasError){
            return (
                <h1>Something Went Wrong with {this.props.children.type.name}!
<br/> Please contact the admin</h1>
             );
        }
        return this.props.children;
    }
}

export default ErrorBoundary;
```

Summary

React follows a proper path of calling all lifecycle methods, which are called in different phases of a component's lifetime. Starting before the component is created, when the component is mounted, updated, or unmounted, and finally, during error handling. They all have roles, but some are used more often than others. The only required method in the whole lifecycle is the render() method.



Summarising it

Let's summarise what we have learned in this Lecture:

- Learned about the component lifecycle.
- Learned about different phases of a lifecycle.
- Learned about lifecycle methods in different phases.
- Learned about the order in which lifecycle methods are called during execution.
- Learned about how and where to perform side effects.

Some References:

- Component Lifecycle: link
- React Lifecycle Methods: link