React Component Lifecycle

 (https://github.com/learn-co-curriculum/react-hooks-class-component-lifecycle)  (https://github.com/learn-co-curriculum/react-hooks-class-component-lifecycle/issues/new)

Learning Goals

- Describe the three phases of the React component lifecycle
- Define lifecycle methods and how they enable the component to react to different events

Overview

In this lesson, we'll describe the phases, as well as the importance, of the React component lifecycle. We'll also talk about the similarities between component lifecycle in class components, and the `useEffect` hook in function components.

Component Lifecycle

React components have two sets of properties: **props** and **state**. Props are given to the component by its parent. You can think of props as an external influence that the component has no control over, whereas a component's state is internal to the component. A component's state can change in conjunction to the props changing or when the user interacts with the component.

The React framework was designed to enable developers to create complex and highly reactive UIs. This enables the components to quickly adapt to changes from user interactions or updates in the app. In order to enable this, React components go through what we call a **component lifecycle**. This is broadly divided into three parts: **creation**, **updating**, and **deletion**.

This means that every single thing you see in applications written in React is actually a React component and/or a part of one! For example, if you open a new chat window in a website written in React, a `ChatWindow` component is **created**. As you are interacting with it and sending messages to your friends — that's the **updating** part. And when you finally close the window, the React component gets **deleted**.

It seems all pretty straightforward from the user's perspective, however as you'll soon find out, there's a lot of stuff going on behind the scenes.

Lifecycle Methods and Rendering

In order to enable this quick reacting and updating, as a developer, you get access to certain built-in events in the React component lifecycle called **lifecycle methods**. These are opportunities for you to change how the component reacts (or doesn't react) to various changes in your app.

These methods are called *lifecycle* methods, because they are called at different times in the component's lifecycle — just before it's created, after it's created, and when it's about to be deleted.

The only required method for a React class component to be valid is the `render()` method, which describes what the HTML for the component looks like. There are a whole host of optional methods you can use if you need more control over how the component responds to change. The optional methods will be called if you include definitions for them in a component. Otherwise, React will follow its default behavior.

There are more lifecycle methods than the ones described below; however, we'll be covering the most common ones. Check out the **React Docs (https://reactjs.org/docs/react-component.html#the-component-lifecycle)** for a full list.

Pre-mounting

It is important to remember that class components, at their core, are just JavaScript classes. This means that even before mounting has begun, the class's `constructor` function is called.

While the `constructor` is not related to mounting to the DOM, it is the first function called upon the initialization of a component.

In older React code, you'll often encounter the `constructor` method when initializing state and binding methods, like this:

```
class App extends React.Component {
  constructor(props) {
    super(props);
    state = {
      count: 0,
    };
```

```
    this.handleClick.bind(this);
  }
}
```

In newer React codebases, you are less likely to encounter the constructor method.

Mounting

When the component is initially created, it gets mounted onto the DOM. "Mounting" for React means getting the component's JSX (whatever is returned by the `render` method), and creating DOM elements based on that JSX.

After the mounting stage, there is one commonly used *lifecycle method* available to us: `componentDidMount`.

The `componentDidMount` method will get called just *after* the `render` method. You would use this method to set up any long-running processes or asynchronous processes such as fetching and updating data. It is better to render and display *something* to your user even if all of your data isn't ready yet. Once it *is* ready, React can just re-render and use the API content.

The `componentDidMount` lifecycle method is *roughly* equivalent to using the `useEffect` hook in a function component with an empty dependencies array. In both cases, we have a place to run some code immediately after the component has been rendered to the DOM. Here's a quick side-by-side:

- **Function Component**:

```
// 1. function is called
function ChatWindow() {
  const [messages, setMessages] = useState([])

  useEffect(() => {
    // 3. side effect function is called
    fetch("http://localhost:3000/messages")
      .then(r => r.json())
      .then(messages => setMessages(messages))
  }, [])
```

```
  // 2. function returns JSX
  return (
    // JSX
  )
}
```

- **Class Component**:

```
class ChatWindow extends React.Component {
  // 1. constructor is callled (state is initialized)
  constructor(props) {
    super(props)
    state = {
      messages: []
    }
  }

  // 3. componentDidMount is called (fetch some data)
  componentDidMount() {
    fetch("http://localhost:3000/messages")
      .then(r => r.json())
      .then(messages => this.setState({ messages }))
  }

  // 2. render is called (return JSX)
  render() {
    return (
      // JSX
    )
  }
}
```

Updating

Whenever a component's state or props are changed, it gets re-rendered on the page. That's the beauty of React components — they're quick to *react* to changes. A re-render could be triggered when a user interacts with the component, or if new data (props or state) is passed in.

For example, going back to the chat window example, whenever you press "send" on a message, the `ChatWindow` component gets re-rendered as it needs to display an extra message. Whenever a re-render is triggered, `render` is called, returning the JSX for React. React uses this JSX to figure out what to display on the page.

After render is called, the `componentDidUpdate` method is called just after the component is rendered and updated. It is possible in `componentDidUpdate` to take some actions without triggering a re-render of the component, such as focusing on a specific form input.

You will have access to the previous props and state as well as the current ones, and you can use this method to update any third party libraries if they happen to need an update due to the re-render.

Comparing to our hooks examples, `componentDidUpdate` can also be used in a similar way to `useEffect` to trigger side effects in response to changes to a component's state or props. Here's an example of two versions of a component with similar functionality to demonstrate:

- **Function Component**

```
// 1. function is called
// 4. function is called again after setting state in useEffect
function ChatRoom() {
  const [roomId, setRoomId] = useState(1);
  const [messages, setMessages] = useState([]);

  useEffect(() => {
    // 3. useEffect callback is called after initial render
    // 6. useEffect callback is called again if roomId changes
    fetch(`http://localhost:3000/rooms/${roomId}/messages`)
      .then((r) => r.json())
      .then((messages) => setMessages(messages));
  }, [roomId]);

  // 2. function returns JSX
  // 5. function returns JSX
```

```
  return; // JSX
}
```

- **Class Component**

```
class ChatRoom extends React.Component {
  // 1. constructor is called
  // (using class field syntax, we don't need to write the constructor method)
  state = {
    messages: [],
    roomId: 1,
  };

  // 3. componentDidMount is called
  componentDidMount() {
    fetch(`http://localhost:3000/rooms/${this.state.roomId}/messages`)
      .then((r) => r.json())
      .then((messages) => this.setState({ messages }));
  }

  // 5. when setState is called, componentDidUpdate runs after render
  componentDidUpdate(prevProps, prevState) {
    if (prevState.roomId !== this.state.roomId) {
      fetch(`http://localhost:3000/rooms/${this.state.roomId}/messages`)
        .then((r) => r.json())
        .then((messages) => this.setState({ messages }));
    }
  }

  // 2. render is called
  // 4. render is called again after setting state in componentDidMount
  render() {
    return; // JSX
```

```
    }
  }
```

As you can see, in the example above there is some duplication of the logic in the `componentDidMount` and `componentDidUpdate` functions — this is one of the advantages of using `useEffect` instead.

Unmounting

At the unmounting stage, the component gets deleted and cleared out of the page. The only lifecycle hook at this stage is `componentWillUnmount`, which is called just before the component gets removed. This is used to clean up after the component by removing timers, unsubscribing from connections, etc.

For example, if you had a component that displays the weather data in your home town, you might have set it up to re-fetch the updated weather information every 10 seconds in `componentDidMount`. When the component gets deleted, you wouldn't want to continue doing this data-fetching, so you'd have to get rid of what was set up in `componentWillUnmount`.

The closest equivalent to `componentWillUnmount` in function components is the cleanup function that you can return from `useEffect`. They aren't *strictly* equivalent, because the cleanup function from `useEffect` will run between each render, and `componentWillUnmount` will only run when the component is removed. But they are both used in similar cases. For example:

- **Function Component**

```
// 1. function is called
// 4. function is called again after setting state
function Timer() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    // 3. useEffect callback runs (only once, since we gave an empty dependencies array)
    const interval = setInterval(() => {
      setCount((count) => count + 1);
    }, 1000);
```

```
    // 6. cleanup function is called after the component is removed (by the parent component)
    return function () {
      clearInterval(interval);
    };
  }, []);

  // 2. function returns JSX
  // 5. function returns JSX
  return; // JSX
}
```

- **Class Component**

```
class Timer extends React.Component {
  // 1. constructor is called
  state = { count: 0 };

  // 3. componentDidMount is called
  componentDidMount() {
    this.interval = setInterval(() => {
      this.setState((prevState) => ({
        count: prevState.count + 1,
      }));
    }, 1000);
  }

  // 5. componentWillUnmount is called after the component is removed (by the parent component)
  componentWillUnmount() {
    clearInterval(this.interval);
  }

  // 2. render is called
  // 4. render is called again after setting state
  render() {
```

```
      return; // JSX
    }
  }
```

Conclusion

Imagine a big old oak tree. The tree could be a parent component, each of its branches a child component of the tree, each of its leaves a child component of the branch and so on. Each of the leaves go through a very obvious lifecycle of being created, changing based on state (changing color based on the season, withering if there's not enough nutrition being passed down from the parent branch, changing into a leaf with a hole bitten out of it if a caterpillar munches on it), and finally falling down when it's autumn.

So as it seems, if you need a lifecycle hook, there's sure to be one for your every need!

Resources

- **Component Lifecycle Methods (https://reactjs.org/docs/react-component.html#the-component-lifecycle)**
- **Component Lifecycle Diagram (https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/)**
- **Understanding the React Component Lifecycle (http://busypeoples.github.io/post/react-component-lifecycle/)**