MODULE: 10 List and Hooks

Explain Life cycle in Class Component and functional component with Hooks

React Hooks and the component lifecycle

Versions of React before 16.8 consider two kinds of components based on statefulness: the class-based stateful component, and the stateless functional components (often referred to as a "dumb component"). But with the release of React 16.8, Hooks were introduced and empowered developers to access state from functional components, instead of writing an entire class. With this change, building components became easier and less verbose.

Hooks known as default hooks come with React, and you're also able to create your own custom hook. A custom hook is just a function that starts with `use`, like `useStore`, or `useWhatever`.

The two most common default hooks are `useState` and `useEffect`. The `useState` hook gives state to the functional component, and `useEffect` allows you to add side effects within it (like after initial render), which aren't allowed within the function's main body. You can also act upon updates on the state with `useEffect`.

React has released more default hooks, but `useState` and `useEffect` are the ones you should be most familiar with. Let's look at how they work and compare them to the component lifecycle we covered above.

`useState`

The `useState` hook is used to store state for a functional component. This hook accepts one parameter: `initialState`, which will be set as the initial

stateful value, and returns two values: the stateful value, and the update function to update the stateful value. The update function accepts one argument, `newState`, which replaces the existing stateful value.

Let's say you have this class-based component:

```
class MyInput extends React.Component {
  constructor(props) {
    super(props);
    this.state = { input: "" };

    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
    this.setState({ input: e.target.value });
  }

  render() {
    return (
      <input
        type="text"
        value={this.state.input}
        onChange={this.handleChange}
      />
    );
  }
}
```

Converting this to a functional component with `useState` eliminates a lot of code and makes things cleaner and shorter. Here's what the above component looks like with the `useState` hook. If you've read a React tutorial over the past 2 years, chances are you've seen some syntax like this.

```
import { useState } from 'react';


function MyInput(props) {
     const [input, setInput] = useState('');

     return (
          <input
               value={input}
               onChange={e => setInput(e.target.value)}
          />
     )
}
```

With `useState()`, whatever you put in the parenthesis is the default state.

The simplicity and clarity of these functional components with Hooks popularized their use among developers who prefer using functional components rather than traditional, class-based ones.

But what if your class-based component's state object has plenty of items like this:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);

    // this component has multiple items in the state object
    this.state = {
      count: 0,
      counterWeight: 1,
      themeMode: "light"
    };

    // ...
```

```
  }

  // ...

  render() {
    return (
      <div class={this.state.themeMode}>
          ...
      </div>
    );
  }
}
```

You might be tempted at first to use one `useState` hook, and build some sort of dictionary to hold all of this state. But React allows you to have multiple `useState` hooks within a functional component. So instead of one single object, you can set each item of the state object into its own state:

```
function MyComponent(props) {
     const [count, setCount] = useState(0);
     const [counterWeight, setCounterWeight] = useState(1);
     const [themeMode, setThemeMode] = useState('light');

     // the rest of the code...

     return (
          <div>
               ...
          </div>
     )
}
```

With a lot of different “types” of state this can obviously get verbose, so exercise judgment for larger components.

`useEffect`

As with the `render()` method of a class-based component, the main body of a functional component should be kept pure. With the `useEffect` hook, you're able to create side effects while maintaining the component's purity. Within this hook, you can send network requests, make subscriptions, and change the `state` value.

The `useEffect` hook accepts a function as an argument, where you write all your side effects. This function is invoked after every browser paint and before any new renders (this will depend on the dependency array, which is explained in the next paragraph). This function can return another function called the clean-up function, which can be used to clean up the side effects (i.e., when the component is destroyed) like unsubscribing to a store. It’s kind of a mash up of several of the methods explained in the previous section.

This Hook accepts a second argument called the dependency array, which is an array of dependencies like `state` or `props` values, which the `useEffect` uses as reference to only run when these values change. If the dependency array is empty, then the `useEffect` will only run once, after the first paint.

The dependency array is optional, so if it's not defined, `useEffect` will fire first when the component is first mounted, and then on every re-render.

Here's an example of a functional component using `useEffect` that subscribes to a Redux store:

```
import { useEffect } from 'react';

function MyComponent(props) {
    // ...
```

```
    useEffect(() => {
        const { subscribe } = props.store;
        const unsubscribe = subscribe(...);
        return unsubscribe
    }, []);

    return
}
```

This component's `useEffect` will only run once, since the dependency array is empty. Within the input function, the subscription to the Redux store is invoked, which returns an `unsubscribe` function. This `unsubscribe` function is returned, which serves as the clean-up function.

You might have noticed that the `useEffect` from the above component has similarities to `componentDidMount`. It holds the side effects, and only runs once, when the component is mounted. The only difference is that it's invoked after the first browser paint, whereas `componentDidMount` doesn't wait for that. It's also important to note that the clean-up function can be compared to the `componentWillUnmount`, as this function is invoked when the component was destroyed. So again, `useEffect()` is a sort of hybrid in this sense.

The `useEffect` hook works similarly to the three lifecycle methods: `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`. The `componentDidMount` and `componentWillUnmount` were discussed above, but what about `componentDidUpdate`?

If you add dependencies in the dependency array, the function passed into the `useEffect` hook will run every time the dependencies, like a piece of stateful data, changes. This behavior of the `useEffect` hook is comparable to the `componentDidUpdate` method since it's invoked on every state/props

change. The difference is that you can specifically choose what state/props you want `useEffect` to depend on, rather than the `componentDidUpdate` which acts upon every `state` or `props` change.

Conclusion

In React, a component can enter three different phases that make up its lifecycle. These phases are mounting, updating, and unmounting. Each phase has lifecycle methods invoked, where you can work on different things related to the component, like its `props` and `state`, or actually mounting the component to the DOM (with the render method). However, these methods are only for class-based components.

With the release of React 16.8, developers can now write stateful functional components with Hooks, which eliminates a lot of verbosity in a class-based component and makes the code easier and simpler to write and read. The two most commonly used default hooks are `useState` and `useEffect` which are used to handle stateful data in a functional component and for creating side effects within. `useEffect` works similarly to the three lifecycle methods: `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`.