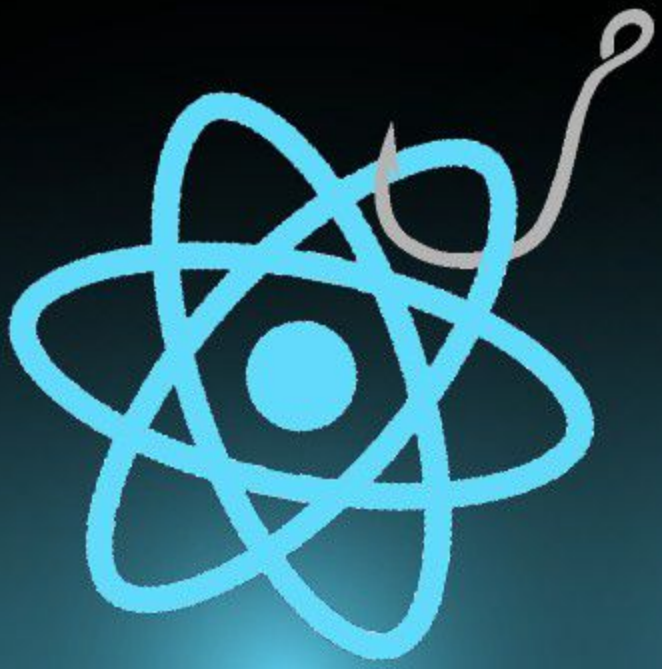
swipe

Introducing

REACT HOOKS



What are Hooks anyway?

swipe

React Hooks let you use state, and other React features without having to define a JavaScript class. It's like being able to take advantage of the cleanliness and simplicity of a Pure Component and state and component lifecycle methods.

A side by side comparison of what the code looks like with and without Hooks for a simple counting component:

```
import './App.css';
import React, { useState } from 'react';

const HooksExample = () => {
    const [counter, setCount] = useState(0);

    return (
        <div className="App">
            <header className="App-header">
                The button is pressed: { counter } times.
                <button
                    onClick={() => setCount(counter + 1)}
                    style={{ padding: '1em 2em', margin: 10 }}
                >
                    Click me!
                </button>
            </header>
        </div>
    )
}

export default HooksExample;
```

What are Hooks anyway?

swipe

NoHooks.js:

```
import './App.css';
import React, { Component } from 'react';

export class NoHooks extends Component {
    constructor(props) {
        super(props;
        this.state = {
            counter: 0
        }
    }

    render() {
        const { counter } = this.state;
        return (
            <div className="App">
                <header className="App-header">
                    The button is pressed: { counter } times.
                    <button
                        onClick={() => this.setState({ counter: counter + 1 }) }
                        style={{ padding: '1em 2em', margin: 10 }}
                    >
                        Click me!
                    </button>
                </header>
            </div>
        )
    }
}

export default NoHooks;
```

Let’s take a look at the hook at hand — useState.

swipe

useState

useState, as the name describes, is a hook that allows you to use state in your function. We define it as follows:

```
const [ someState, updateState ] = useState(initialState)
```

Let’s break this down:

- someState: lets you access the current state variable, someState

- updateState: function that allows you to update the state whatever you pass into it becomes the new someState

- initialState: what you want someState to be upon initial render

So we've got state in hooks. What about component lifecycle methods?

useEffect

useEffect is another hook that handles componentDidUpdate, componentDidMount, and componentWillUnmount all in one call. f you need to fetch data, for example, you could useEffect to do so, as seen below.

```
import React, { useState, useEffect } from 'react';
import axios from 'axios';

const HooksExample = () => {
    const [data, setData] = useState();
    useEffect(() => {
        const fetchGithubData = async (name) => {
            const result = await axios(`https://xx/${name}/x`)
            setData(result.data)
        }
        fetchGithubData('xx')}, [data])
    return (
        <div className="App">
            {data && (data.map(item => <p>{item.repo.name}</p>))}
        </div>
    )
}
export default HooksExample;
```

swipe

So we've got state in hooks. What about component lifecycle methods?

useEffect

Taking a look at useEffect we see:

- First argument: A function. Inside of it, we fetch our data using an async function and then set data when we get results.

- Second argument: An array containing data. This defines when the component updates. As I mentioned before, useEffect runs when componentDidMount, componentWillUnmount, andcomponentDidUpdate would normally run. Inside the first argument, we've set some state, which would traditionally cause componentDidUpdate to run. As a result, useEffect would run again if we did not have this array. Now, useEffect will run on componentDidMount, componentWillUnmount, and if data was updated, componentDidUpdate. This argument can be empty— you can choose to pass in an empty array. In this case, only componentDidMount and componentWillUnmount will ever fire. But, you do have to specify this argument if you set some state inside of it.

useReducer

For those of you who use Redux, useReducer will probably be familiar. useReducer takes in two arguments — a reducer and an initial state. A reducer is a function that you can define that takes in the current state and an “action”.

We can pass this reducer into useReducer, and then use this hook like this:

```
const [ state, dispatch ] = useReducer(reducer, initialState)
```

You use dispatch to say what action types you want to execute, like this:

```
dispatch({ type: name})
```

useReducer

useReducer is normally used when you have to manage complex states

```
import React, { useReducer } from 'react';

const reducer = (state, action) => {
    switch (action.type) {
        case 'firstName': {
            return { ...state, firstName: action.value };
        }
        default: {
            return state;
        }
    }
};
function SignupForm() {
    const initialState = {
        firstName: '',
    }
    const [formElements, dispatch] = useReducer(reducer, initialState);
    return (
        <div className="App">
            <input placeholder="First Name" value={formElements.firstName}
            onChange={(e) => dispatch({ type: firstName, value: e.target.value })} />
        </div>
    );
}
export default SignupForm;
```

useContext

This hook allows you to work with React context API which is a mechanism used to share values throughout the component tree (share data without passing props).

Let's say for example that we have an object called setting that contain two values for user setting:

```
const settings = {
    notification: true,
    newsletter: false
}
```

To share these setting across multiple disconnected components we can create a context:

```
const settings = {
    notification: true,
    newsletter: false
}

const SettingsContext = createContext(settings);
```

useContext

If we want to scope the setting in a part of our application we will use a context provider and any child component will inherit that value without passing any "props":

```
import { SettingsContext } from "twilio/lib/rest/voice/v1/dialingPermissions/settings";

function App() {
    return (
        <SettingsContext.Provider value={SettingsContext.notification}>
            <SomeOtherComponents />
        </SettingsContext.Provider>
    )
}
```

And here appears useContext hook which allows us to consume the current value from the context provider which may live many levels higher in the component tree.

```
function MyComponent() {
    const setings = useContext(SettingContext);
    return (
        // code...
    )
}
```

swipe

useRef

This hook will allow you to create an immutable object to keep the same reference between renders.

A more common use case for useRef is grabbing html elements from the DOM.

```
function App() {
    const myBtn = useRef(null);
    const clickProgramatically = () => myBtn.current.click()
    return (
        // connect the ref to an html button using ref attribute
        <button ref={myBtn}></button>
    )
}
```

swipe

useMemo

This hook helps in optimizing computation cost to improve performance, it's commonly used with expensive computations.

Instead of computing in every render we can memoize the value. We write a function that returns the computed value with dependencies as a second argument to determine when this computation should be run.

```
function App() {
    const [count, setCount] = useState(100);
    const expensiveCount = useMemo(() => {
        return count ** 8;
    }, [count])
    //...
}
```

swipe

useCallback

When you define a function in a component a new function object is created every time the component is rerendered, in some cases you may also want to memoize the function.

A common use case when you pass the same function to child components. By wrapping the function in useCallback we can prevent unnecessary rerenders of the children because they are using the same function object.

```
function App() {
    const [count, setCount] = useState(100);
    const showCount = useCallback(() => {
        alert(`Counter ${count}`);
    }, [count])
    return (
        <ChildComponent handler={showCount} />
    )
}
```

Custom Hooks

Custom hooks must start with use

```
function useApiResult = (param) => {
    const [result, setResult] = useState(null);
    useEffect(() => {
        // your task
    }, [param]);
    return { result };
};

// To use it in a component:
const { result } = useApiResult('some-param');
```