Web Front-End Development
Week 12: React Hooks

React Hooks

https://reactjs.org/docs/hooks-intro.html

Hooks aim to solve a couple of problems in React

- Stateful logic is hard to reuse between components
  - Creating higher-order components to lift state up requires restructuring the component hierarchy
- Complex components are hard to understand
  - Unrelated stateful logic and lifecycle logic get mixed together in lifecycle methods
- Classes are the only components that can be stateful and they're confusing
  - Hooks lets you use state without the need for classes, the keyword this, and binding event handler functions.

Hooks were added into React 16.8 which was released 2/6/19. They are additional functionality so they don't change or replace anything already in React.

Hooks are functions that let you "hook into" React state and lifecycle events from components that are functions and not classes.
React provides a few built-in hooks.

State Hook

https://reactjs.org/docs/hooks-state.html

The state hook can be used in a function component replacing `this.state` in a class component.
`useState` declares a state variable and takes as a parameter its initial value

- Does not need to be an object as it does in a class component (although it can be)

`useState` returns a pair of values – an array with two items

- The first item is the current state
- The second is a function that lets us update the state

`const [count, setCount] = useState(0)` is similar to `this.state.count` and `this.setState` in a class component.
This JavaScript syntax is called "array destructuring". It means that we're making two new variables `count` and `setCount`, where `count` is set to the first value returned by `useState`, and `setCount` is the second. It is equivalent to this code:

```
var countStateVar = useState(0); // Returns a pair
var count = countStateVar[0]; // First item in a pair
var setCount = countStateVar[1]; // Second item in a pair
```

You can use `useState()` to declare as many state variables as you need.

To read the value you can access it directly in JSX `{count}`. No `this.state` is needed.

To update state you use the function created `setCount(count+1)`. `this.setState()` is not needed. Unlike `this.setState` in a class, updating a state variable always *replaces* it instead of merging it.

Note that function components have no `this` and therefore functions that are event listeners also don't need to use `bind(this)`.

Rules
https://reactjs.org/docs/hooks-rules.html
Only call hooks in function components

- Can't be used in class components
- Don't call them from regular JavaScript functions

Only call hooks at the top level of a function component

- Don't call hooks in loops, conditionals, or nested functions

Hooks can call other hooks

There is an ESLint plugin called eslint-plugin-react-hooks that enforces these two rules. This plugin is included by default in Create React App.

Example
https://repl.it/@aileenjp/react-favorites-hooks

Change Favorite into a function that use hooks instead of a class that defines this.state in the constructor.

Import the `useState` hook from React by adding it to your import statement.
```
import React, { useState }  from 'react';
```

Change
```
class Favorite extends React.Component{}
```
to
```
function Favorite(props){}
```

The constructor is removed and replaced with
```
const [votes, updateVotes] = useState(0);
```

Replace
```
{this.state.votes}
```
with
```
{votes}
```

To update votes instead of calling setState()
```
    this.setState({
      votes: this.state.votes +1
    });
```
call

```
updateVotes(votes + 1);
```

All occurences of `this.props` are replaced with `props`.
`this.handleClick is` replaced with `handleClick`

Remove the call to `render()`.

Effect Hook
https://reactjs.org/docs/hooks-effect.html
The Effect Hook adds the ability to perform operations/side effects from a function component.

- Setting up a subscription
- Data fetching
- Network requests
- Similar to `componentDidMount(), componentDidUpdate()`, and `componentWillUnmount()` in React classes, but unified into a single API

Effects are declared inside function components so they have access to props and state.
`useEffect` is React's hook to perform operations associated with changes to the DOM

- define a function to be run after every change to the DOM
    - React runs effects after every render including the first render by default
- you can optionally return a function to handle cleanup when the component unmounts
    - return a function from `useEffect` to be the cleanup function
    - the cleanup function is also called before subsequent renders in case data, such as props, has changed

```
function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });
}
```

In this example we're updating the document's title after each render

- count is a state variable
- we pass a function to the `useEffect` hook
- the function we pass to the `useEffect` hook is the effect
- has access to state (and props if there were any)
- our effect function will be run after every render

You can declare as many effects using `useEffect()` as you need in order to separate unrelated logic.

`useEffect()` handles updates automatically but to optimize performance you can also tell React to skip applying an effect if certain values haven't changed between re-renders.

- Pass an array as an optional second argument to `useEffect()`
- React will use the value from the array and compare the value in the previous render to the value from the next render and if they're the same React will skip the effect

Example

Instead of using the lifecycle methods such as `componentDidMount()` and `componentDidUpdate()` we'll add an effect so we can see how it's automatically called after each render.

Import `useEffect`

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

Add an effect

```
  useEffect(() => {
    console.log("This is an effect");
  });
```

Update return so we can see when the component is being rendered.

```
{console.log("In render")}
```

Open in separate window to view the console.

There are other built-in hooks available as well https://reactjs.org/docs/hooks-reference.html or build your own.

Build your own Hooks

https://reactjs.org/docs/hooks-custom.html

You can also build your own hooks which is helpful when you want to reuse stateful logic between components, eliminating the need to add more components to your hierarchy.

Custom hooks are regular JavaScript functions

- can also call other hooks, just make sure you do this at the top level of your custom hook
- can take arguments if needed
- can return data if needed
- name should start with "use"

Multiple components can use a custom hook so the state, and the state's logic, is reused. You are not "sharing" state, there is still the single source of truth for the state, which is the custom hook.