React Hooks I

INFO 253A: Frontend Web Architecture

Kay Ashaolu

What are Hooks?

- Hooks are a way to be able to use the full functionality of React in a more functional programming kind of way
- You no longer require JavaScript classes to write fully fledged React Components
- I have not taught ES6 classes nor React Class components because of this move

Why?

- In my opinion, React have always embraced the encapsulating properties of a function conjoined with it's simplicity.
- The concept of having a component that does not depend on anything else marries well with the function concept of the result being dependent on the inputs of the function
- **The abstraction of a function can be elegantly used in this context**

But why hooks?

- Functional components already did exist in React before hooks
- However they were limited in what they can do
    - They can accept properties and render HTML based on their properties
    - But they couldn't tie into some of the more fundamental and advanced features of React
    - Class based components could define special functions that had special properties that "hooked" into React functionality
- But with hooks, functions also have the same ability

Let's start with an example: useState

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

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

useState example

- useState is a Hook
- the useState function returns two elements: a current state value and a function that enables you to update
- useState takes a single argument: the initial state value.

Now what really is an Hook?

- Hooks are functions that allow you "hook into" React features like state and what's called "lifecycle" features from function components
- An example of a lifecycle feature is
  - Execute code when component is first created
  - Execute code when component updates

What is state in React?

- A state variable is a single piece of data that resides within each component
- Each instance of the component "remembers" its own state
- When any state variable is changed, React re-renders the component, incorporating any changes

Let's go back to our example

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

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

What's happening here?

- In our Example component, we set a single element of state called count
- We have access to the current value of count using the "count" variable, and the function "setCount" that takes one parameter (future state) that can change the count variable
- count, and setCount are declared as const to declaratively state that they cannot be changed. You cannot change count by assigning it to another value. But you must use the setCount function to change the count value
- Using the setCount function is important: when state is changed using this function React knows to render the component again after the variable has changed

What's happening here?

- Because the button's onClick attribute is set to an anonymous function that increments count (using the setCount function), the component is rendered again with the new value of the button

useEffect hook

- The useEffect hook gives you access to what's called React's "lifecycle features"
- Lifecycle features in this case means access to special times in the creation, operation, and removal of a componnet.
- useEffect state takes one function that will be executed right after the component is rendered to the screen.
- In effect, this hook gives you the ability to run code on startup, and when any state changes in the component, and when the component is removed

useEffect Example

```javascript
useEffect(() => {
  const subscription = props.source.subscribe();
  return () => {
    // Clean up the subscription
    subscription.unsubscribe();
  };
});
```

useEffect Example

- useEffect here is passed a function that contains two statements:
- First, it is subscribing to whatever props.source.subscribe() is. This will be done any time this component is rendered to the screen
- Second, if this component is removed, then the function that is returned will execute (the unsubscribe action)
- This function it is returning enables you to clean up any actions that may not be needed anymore

Grand Example

```javascript
import React, { useState, useEffect } from 'react';
import ReactDOM from 'react-dom';

function Weather(props) {

  const [temp, setTemp]  = useState(0);

  let getWeatherData = async () => {
    let response = await fetch(`https://api.openweathermap.org/data/2.5/weather?

    if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
        } else {
                return response.json();
  }

}

  useEffect(() => {

    getWeatherData().then((response) => {
      setTemp(response.main.temp);
    }).catch(e => console.log(e));;
  })

```

Grand Example

```jsx
return (
    <div>
      <strong>The temperature of {props.location} is {temp}</strong>
    </div>
  );

}

function App(props) {
  return (
    <div>
      <Weather location="Berkeley,ca" />
      <Weather location="Concord,ca" />
    </div>
  );
}

ReactDOM.render(
<App />,
document.getElementById('root')
);
```

Grand Example

- Note we are using both useState to keep the state of the temperature and useEffect to make an API call to the weather endpoint to get the weather
- The function in useEffect is executed on every render, but since we only pass a property of the current location, it only needs to be rendered once

Questions?