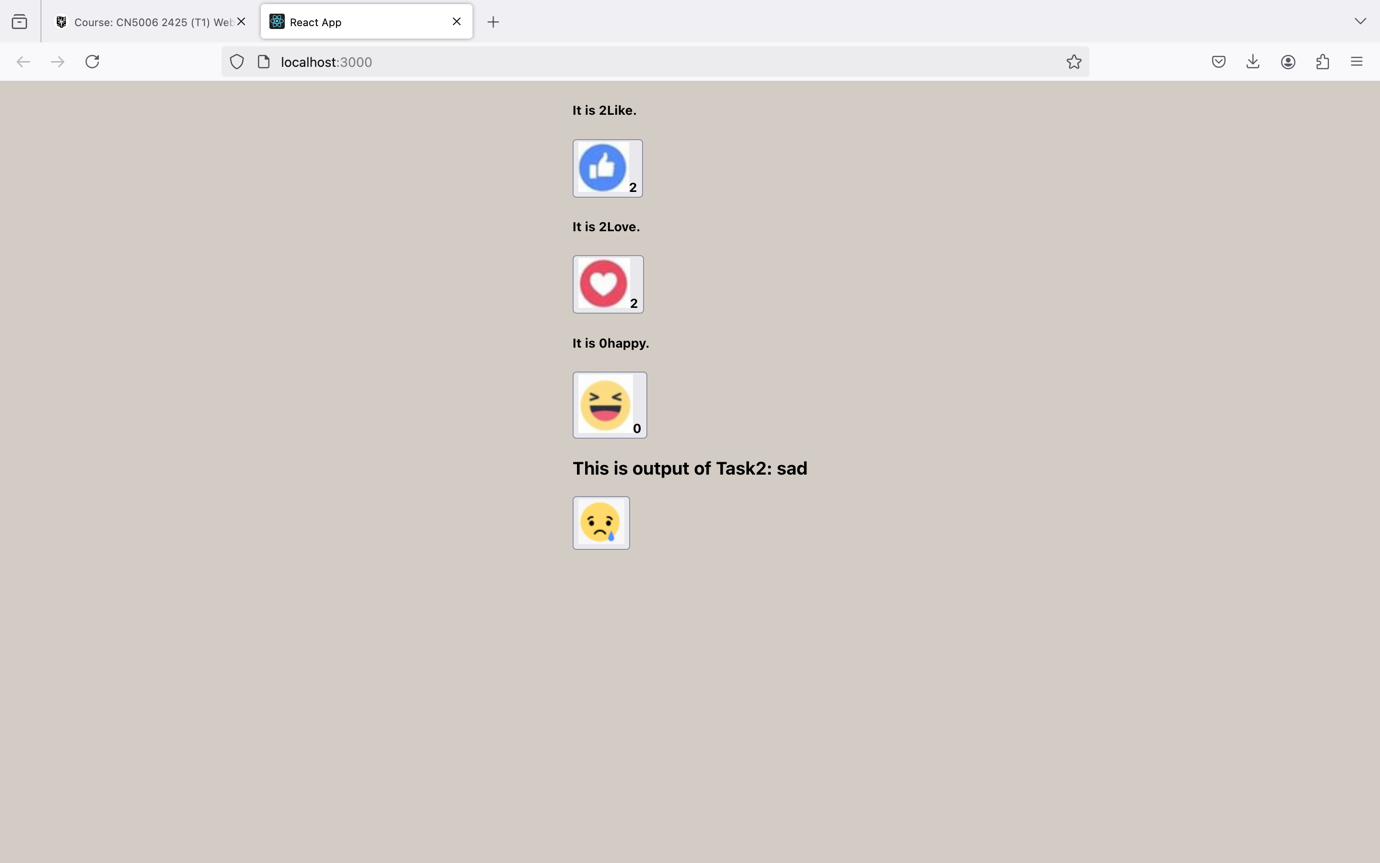
Week 9



**Reflection on Functional vs. Class Components**

**Examples Overview**

1. **toggle.js (Class Component)**:
   * This file demonstrates a class-based component named ToggleMode, which toggles between two images (happy and sad) when a button is clicked. The component maintains its state (pic) using the state property and updates it via the setState method.
2. **emoji.js (Class Component)**:
   * This file showcases another class-based component named FacebookEmojiCounter. It increments a counter displayed alongside an emoji (Like, Love, or Happy) based on the type prop. The emoji image is determined conditionally in the constructor, and the state (number) increments with a button click.

**Differences Between Class Components and Functional Components**

1. **State Management**:
   * **Class Components**: Use the state object and setState method to manage state. This approach is explicit but requires more boilerplate, such as defining a constructor and binding methods.
   * **Functional Components**: Utilize hooks like useState, which are simpler and more intuitive. With hooks, there's no need for constructors or binding.

Example for ToggleMode in a functional component:

import React, { useState } from "react";

import sad from "./sad.png";

import happy from "./happy.png";

const ToggleMode = () => {

const [pic, setPic] = useState(happy);

const mode = pic === happy ? "happy" : "sad";

const toggleMode = () => {

setPic((prevPic) => (prevPic === happy ? sad : happy));

};

return (

<div>

<h3>This is output of Task2: {mode}</h3>

<button onClick={toggleMode}>

<img src={pic} alt=" " />

</button>

</div>

);

};

export default ToggleMode;

2. **Lifecycle Methods**:

* **Class Components**: Require explicit lifecycle methods like componentDidMount and componentDidUpdate.
* **Functional Components**: Leverage the useEffect hook, which consolidates lifecycle behavior into a single, flexible mechanism.

If FacebookEmojiCounter needed a lifecycle behavior (e.g., logging each increment), it would look cleaner in a functional component:

import React, { useState, useEffect } from "react";

import like7 from "./like7.png";

import Love from "./Love.png";

import happy from "./happy.png";

const FacebookEmojiCounter = ({ type }) => {

const [number, setNumber] = useState(0);

const pic = type === "Love" ? Love : type === "Like" ? like7 : happy;

useEffect(() => {

console.log(`Current count is: ${number}`);

}, [number]);

return (

<div>

<h5>It is {number} {type}.</h5>

<button onClick={() => setNumber((prev) => prev + 1)}>

<img src={pic} alt=" " />

<b>{number}</b>

</button>

</div>

);

};

export default FacebookEmojiCounter;

3. **Boilerplate Code**:

* **Class Components**: Require constructors, method binding, and explicit syntax for lifecycle methods, which can be verbose.
* **Functional Components**: Provide a streamlined syntax, reducing code size and complexity.

4. **Readability and Maintenance**:

* **Class Components**: Can become cluttered as the state and lifecycle logic grow, making it harder to maintain.
* **Functional Components**: Encourage cleaner separation of concerns using hooks like useState and useEffect, improving readability.

#### ****Conclusion****

While class components were foundational in React's evolution, functional components with hooks have largely supplanted them in modern development due to their simplicity, flexibility, and readability. The provided examples (toggle.js and emoji.js) would benefit from refactoring to functional components for a cleaner and more modern approach.