**Week 6**

**6.1 React Hooks**

* Component returns a single element, so that it is easy to do reconciliation. Reconciliation is the act of updating the DOM as the application grows.
* Returning <></> gets rid of the extra div. We can also use <React.fragment></React.fragment>
* Writing in { } causes the thing to be destructured. import React from “react” or import {Fragment} from “react”.
* Whenever a state variable inside the component changes, the component re-renders. When a parent re-renders, all the children re-render, irrespective of whether they are using the state variable or not.
* To solve this, push the state down as much as you can. The state should be at the lowest common ancestor. The lowest common component.
* Another way is using React.memo. In this method, the state is at the top most component and the parent always re-renders but the children where the state doesn’t change, they don’t re-render.
* States should always be passed from parent to child and not the other way around.
* Whenever we’re re-rendering a list, make sure that list has a unique identifier and that key is provided when we’re re-rendering. It helps React in calculating diff.
* Wrapper component takes other component as input. It can also take normal xml.
* Wrapper component has access to children. Children are the xml or any other component passed to the wrapper.
* We can have wrappers inside wrappers.
* Anything that starts with a use is called a hook.
* Life cycle events 🡪 mount, unmount, re-renders,
* useEffect() 🡪 when this component mounts, do something. If we give an empty array as the second argument, it means that it would run only when the component is mounted. This array is called the dependency array. If we don’t provide the dependency array, it would run infinitely.
* If we put a state in the dependency array, it means that the function would run whenever the state gets updated.
* You can’t write an async function inside useEffect.

**6.2 Common Hooks**

* Side effects are doing a backend call, setTimeout, setInterval.
* useEffect() lets you put some conditions as to when to run the function.
* You put state variables inside dependency array.
* Axios is better than fetch.
* Across renders, if we want to remember a value then use useMemo.
* When React sees that the passed thing has changed, it re-renders the component always. Even if the same function/object is passed, the component re-renders. Therefore, we use useCallback and wrap the function inside useCallback.
* memo is different from useMemo. If we wrap a component inside a memo, then it doesn’t re-render when the props are unchanged.
* useEffect() 🡪 when to re-render/re-render when something changes.
* useMemo() 🡪 re-render some lines of code when something changes/remembers the original code, memoize it. Using useEffect() causes an extra state variable and an extra rendering. You put the code inside a function and then the function inside useMemo().
* useCallback() 🡪 does not re-render component when the inputFunction does not change/memoize the function signature.
* Custom hooks should start with a use.
* Whenever we want to do side effects, we put them inside useEffect() 🡪 setTimeout, setInterval, fetch, etc.
* useMemo() is for memoizing values, useCallback() is for memoizing functions.

**6.3 React Recap**

* Reconciliation means writing things on the DOM.
* ReactDOM when used with React makes changes on the website.
* Re-rendering is updating the DOM.
* Getting data from backend, setInterval, and setTimeout are all side effects.
* useEffect() 🡪 dependency array.
* When a component re-renders, all lines re-render even when the states used do not change. To solve this, we use useMemo().
* Key difference between useEffect() and useMemo() is that useMemo() returns while useEffect() does not.
* If a parent re-renders all child re-renders.
* React.memo lets you stop re-renders when the props are unchanged. But, when a parent component re-renders and some function in parent component also re-renders, then its signature changes, if that function is passed to the child as a prop then props change and the child component re-renders.
* useCallback() stops re-rendering function, specifically when that function is passed as a prop to the child component. If we use any state variable inside the function, then we have to put that variable inside the dependency array.
* Doing setCount(count => count + 1) instead of setCount(count + 1) helps to prevent use of state variable count. setCount(count => count + 1) === setCount(currentCount => currentCount + 1). We don’t have to put count in the dependency array.
* useRef() is used to overwrite something React has written on the screen. useRef() is used to get reference to DOM elements.
* useRef() is also used when you want to have access to a variable across renders that is not a state variable. We can store integers, strings, etc. using useRef().
* You either use it as a reference and not put anything inside () or you use it as a variable and initialize it with integers, strings, etc.