**Report about**

“ React Hooks ”

React Training – Task #10

The report includes What you learned about React Hooks.

Basic Hooks (useState, useEffect).

Additional Hooks (useCallback, useMemo, useRef, useLayoutEffect, useId).

* **Introduction**

What is a Hook, Hooks allow us to "hook" into React features such as state and lifecycle methods.

Hooks allow function components to have access to state and other React features. Because of this, class components are generally no longer needed.

Although Hooks generally replace class components, there are no plans to remove classes from React.

* **Basic Hooks**
  + - **useState:** Returns a stateful value, and a function to update it.
    - **useEffect:** allows to perform side effects in your components.
* **Additional Hooks**
  + - **useCallback:** returns a memoized callback function.
    - **useMemo:** returns a memoized value.
    - **useRef:** returns a mutable ref object whose .current property is initialized to the passed argument (initialValue).
    - **useLayoutEffect:** The signature is identical to useEffect, but it fires synchronously after all DOM mutations.
    - **useId:** for generating unique IDs that are stable across the server and client, while avoiding hydration mismatches.
* **Subject** (Content)

1. **Basic Hooks**

* **useState**

The React useState Hook allows us to track state in a function component, State generally refers to data or properties that need to be tracking in an application.

Import useState, To use the useState Hook, we first need to import it into our component.

import { useState } from "react";

Initialize state at the top of the function component.

import { useState } from "react";

function FavoriteColor() {

const [color, setColor] = useState("");

}

* **useEffect**

The useEffect Hook allows you to perform side effects in your componentsSome examples of side effects are: fetching data, directly updating the DOM, and timers.

useEffect accepts two arguments. The second argument is optional.

useEffect(<function>, <dependency>)

useEffect runs on every render. That means that when the count changes, a render happens, which then triggers another effect.

This is not what we want. There are several ways to control when side effects run.

We should always include the second parameter which accepts an array. We can optionally pass dependencies to useEffect in this array.

1. No dependency passed:

useEffect(() => {

//Runs on every render

});

2. An empty array:

useEffect(() => {

//Runs only on the first render

}, []);

3. Props or state values:

useEffect(() => {

//Runs on the first render

//And any time any dependency value changes

}, [prop, state]);

4. Effect Cleanup:

useEffect(() => {

const subscription = props.source.subscribe();

return () => {

// Clean up the subscription

subscription.unsubscribe();

};

};

1. **Additional Hooks**

* **useCallback**

The React useCallback Hook returns a memoized callback function.

Think of memoization as caching a value so that it does not need to be recalculated.

This allows us to isolate resource intensive functions so that they will not automatically run on every render.

The useCallback Hook only runs when one of its dependencies update.

This can improve performance.

The useCallback and useMemo Hooks are similar. The main difference is that useMemo returns a memoized value and useCallback returns a memoized function.

* **useMemo**

The React useMemo Hook returns a memoized value, Pass a “create” function and an array of dependencies. useMemo will only recompute the memoized value when one of the dependencies has changed. This optimization helps to avoid expensive calculations on every render.

* **useRef**

The useRef Hook allows you to persist values between renders, It can be used to store a mutable value that does not cause a re-render when updated, It can be used to access a DOM element directly.

useRef returns a mutable ref object whose .current property is initialized to the passed argument (initialValue). The returned object will persist for the full lifetime of the component.

* **useLayoutEffect**

The signature is identical to useEffect, but it fires synchronously after all DOM mutations. Use this to read layout from the DOM and synchronously re-render. Updates scheduled inside useLayoutEffect will be flushed synchronously, before the browser has a chance to paint.

Prefer the standard useEffect when possible to avoid blocking visual updates.

* **useId**

useId is a hook for generating unique IDs that are stable across the server and client, while avoiding hydration mismatches, useId is not for generating keys in a list. Keys should be generated from your data.

* **Conclusion**