**1.)What are controlled / uncontrolled components? Why do we use controlled components?**

**Controlled Components**

In a controlled component, the form data is handled by the state *within the component*. The state within the component serves as *“the single source of truth”* for the input elements that are rendered by the component.

## Uncontrolled Components

Uncontrolled components act more like traditional HTML form elements. The data for each input element is stored in the DOM, not in the component. Instead of writing an event handler for all of your state updates, you use a **ref** to retrieve values from the DOM.

## Controlled vs. uncontrolled components: Key differences

Controlled components are predictable because the state of the form elements is handled by the component.

Uncontrolled components are not predictable because, during the lifecycle of a component, the form elements can lose their reference and may be changed/affected by other sources

Controlled components enable you to effectively employ form validation to your forms. It doesn’t matter what changes the form elements. Their values are safe in our local states, so that’s where we perform our validation

With controlled components, you are very much in control of your form elements’ values. You can dictate how they go and what can and cannot be inserted

**The use of controllers**

For developers that use React, controlled components should be what you use**when handling form data in your applications.** They create a single source of truth, enable in-place feedback, and make unit testing much easier. Though uncontrolled components have their uses, they aren’t exactly the “React Way” and should be moved away from.

**2) What are side effects? Write a note about them.**

Side effects are basically anything that affects something outside of the scope of the current function that’s being executed. In our dashboard, this includes:

* API requests to our backend service.
* Calls to our authentication service.
* Error tracking calls to Sentry.

Side Effect is not a react-specific term. It is a general concept about behaviours of functions. A function is said to have side effect if it tries to modify anything outside its body. For example, if it modifies a global variable, then it is a side effect. If it makes a network call, it is a side effect as well.

Common features of side-effects For example, when a user clicks “Save,” you may want to fire off an AJAX request. Side-effects may dispatch Redux actions. Like when the save process finishes successfully, you may want to dispatch SAVE\_SUCCEEDED ; or when it failed, SAVE\_FAILED . They also may not dispatch anything.

**3) What is the uni-directional data flow in React?**

React is a library that allows developers to build reusable components and insist on one-way data flow. In general, this concept means that data has one, and only one, way to be transferred to other parts of the application. One way data flow is sometimes called **Unidirectional Data flow** or one-way binding. Unidirectional flow data can only be transferred by one way to the other parts of the application generally top to bottom components ( parent component to its nested child components) so a child should never update the data which is coming from its parent and it should be read-only. In React, the data stored in the parent component is called a state and the snapshot of such data which is passed to its children components via read-only is called a prop.

Here are some reason why using Unidirectional data flow is really helpful while developing an application:

Since the state is maintained at a single location it can be easily traced what actions are updating the data as the flow is one way only.

You can control which components to re-render on a common state change. You put all those components under one common state containing the parent component.

Easy to trace and fix bugs and errors caused by any bad code.

**some key advantages**:

* it’s less error prone, as you have more control over your data
* it’s easier to debug, as you know *what* is coming from *where*
* it’s more efficient, as the library already knows what the boundaries are of each part of the system

**4) List down and explain the 3 ways in which useEffect() hook can be used, with relevant examples.**

**What does useEffect do?** By using this Hook, you tell React that your component needs to do something after render. React will remember the function you passed (we’ll refer to it as our “effect”), and call it later after performing the DOM updates. In this effect, we set the document title, but we could also perform data fetching or call some other imperative API.

**Why is useEffect called inside a component?** Placing useEffect inside the component lets us access the count state variable (or any props) right from the effect. We don’t need a special API to read it — it’s already in the function scope. Hooks embrace JavaScript closures and avoid introducing React-specific APIs where JavaScript already provides a solution.

**Does useEffect run after every render?** Yes! By default, it runs both after the first render *and* after every update. (We will later talk about [how to customize this](https://reactjs.org/docs/hooks-effect.html#tip-optimizing-performance-by-skipping-effects).) Instead of thinking in terms of “mounting” and “updating”, you might find it easier to think that effects happen “after render”. React guarantees the DOM has been updated by the time it runs the effects.

Example:

Import React , { useState, useEffect} from ‘react’;

Function Example(){

const [count, setCount] = useState(0);

useEffect(() => {

document.title =’you clicked ${count} times’;

})

Return (

<div>

<p> You clicked {count} times </p>

<button onClick ={() => setCount(count + 1)}>

Click me

</button>

</div>

);

}