### **Common Interview Questions**

**References:** https://github.com/sudheerj/reactjs-interview-questions?tab=readme-ov-file#what-is-jsx

### What is React?

### What are the major features of React?

1. Prop drilling
2. Context - **React Context** is a powerful feature that allows you to share state across multiple components without having to pass props down manually at every level. This can be particularly useful for managing global state, such as user authentication, theme settings, or locale information.
3. Redux( Why, when and how)
4. Redux ToolKit(RTK)
5. Cutom Hook
6. Lazy Loading
7. Virtual DOM (Reconciliation, diffing algorithm, react fiber)
8. Async tasks

# **What is Single Page Application?**

Reference: https://www.geeksforgeeks.org/reactjs-pure-components/

A **Single Page Application (SPA)** is a type of web application that loads and updates content dynamically without refreshing the entire page.

A single-page application is a website that loads a single document and overwrites the existing page with new data from a web server rather than reloading pages individually from scratch. As a result, page content updates in real time based on user actions with quick transitions and without refreshing.

**What is React?**  
React is a JavaScript library for building user interfaces. It's a view layer for web applications that allows you to create reusable UI components. React is designed to simplify the process of creating interactive and dynamic user interfaces for web and mobile applications.

### **What are the rules of JSX?**

The below 3 rules needs to be followed while using JSX in a react application.

* **Return a single root element**: If you are returning multiple elements from a component, wrap them in a single parent element. Otherwise you will receive the below error in your browser console.

*“html Adjacent JSX elements must be wrapped in an enclosing tag*.”

* **All the tags needs to be closed:** Unlike HTML, all tags needs to closed explicitly with in JSX. This rule applies for self-closing tags(like hr, br and img tags) as well.
* **Use camelCase naming:** It is suggested to use camelCase naming for attributes in JSX. For example, the common attributes of HTML elements such as class, tabindex will be used as className and tabIndex.  
  **Note:** There is an exception for aria-\* and data-\* attributes which should be lower cased all the time.

## **What are Pure Components in React?**

In simple words, If the previous value of the state or props and the new value of the[**state**](https://www.geeksforgeeks.org/reactjs-state/) or [**props**](https://www.geeksforgeeks.org/reactjs-props-complete-reference/) are the same, the component will not re-render itself. **Pure Components** restricts the re-rendering when there is no use for re-rendering of the component. Pure Components are Class Components that extend **React.PureComponent**. Understanding Pure Components is key to optimizing performance in React apps

***Key feature of Pure Component***

### **Shallow Comparison:** Pure components perform a shallow comparison of the props and states. If the objects are passed as props or states have the same references, a re-render is prevented.

### **Performance Optimization:** Pure components can provide performance optimizations by preventing unnecessary re-renders when the data is the same and hasn’t been modified.

### **ShouldComponentUpdate:** Pure components automatically implement [shouldComponentUpdate()](https://www.geeksforgeeks.org/reactjs-shouldcomponentupdate-method/) method with a shallow prop and state comparison. This method returns false if the props and state haven’t changed.

**How useCallback works**

In React, the useCallback hook is used to optimize performance by memoizing callback functions. This helps prevent the recreation of functions on every render.

1. **Basic Usage**: The useCallback hook takes two arguments:
   1. The first argument is the function you want to memoize.
   2. The second argument is an array of dependencies that determine when the function should be recreated.

const memoizedCallback = useCallback(() => {  
 *// Your function logic here*  
}, [dependency1, dependency2]);

1. **Memoization**: When the component renders, useCallback checks if any of the dependencies have changed since the last render. If none of the dependencies have changed, useCallback returns the same function instance from the previous render. If any of the dependencies have changed, it creates a new function instance.

**What is Strict mode and its benifits?**

React.StrictMode is a useful component for highlighting potential problems in an application. Just like <Fragment>, <StrictMode> does not render any extra DOM elements. It activates additional checks and warnings for its descendants. These checks apply for *development mode* only.

This will be helpful in the below cases,

1. To find the bugs caused by impure rendering where the components will re-render twice.
2. To find the bugs caused by missing cleanup of effects where the components will re-run effects one more extra time.
3. Identifying components with **unsafe lifecycle methods**.
4. Warning about **legacy string ref** API usage.
5. Detecting unexpected **side effects**.
6. Detecting **legacy context** API.
7. Warning about deprecated **findDOMNode** usage

### **Why does strict mode render twice in React?**

StrictMode renders components twice in development mode(not production) in order to detect any problems with your code and warn you about those problems. This is used to detect accidental side effects in the render phase. If you used create-react-app development tool then it automatically enables StrictMode by default.

### **What is the difference between useState and useRef hook?**

* useState causes components to re-render after state updates whereas useRef doesn’t cause a component to re-render when the value or state changes. Essentially, useRef is like a “box” that can hold a mutable value in its (.current) property.
* useState allows us to update the state inside components. While useRef allows referencing DOM elements.

## **What Are Closures?**

A closure in JavaScript is created when a function is defined within another function. It allows the inner function to access the variables and parameters of the outer function, even after the outer function has finished executing.

In simpler terms, a closure allows a function to access variables from its outer scope even after that scope has closed.

**What is promise**

A promise is an object returned by an asynchronous function, which represents the current state of the operation. At the time the promise is returned to the caller, the operation often isn't finished, but the promise object provides methods to handle the eventual success or failure of the operation.

**Higher Order Components**

* A higher-order component (HOC) is an advanced technique in React for reusing component logic.
* A higher-order component is a function that takes a component and returns a new component.

Reason to use Higher-Order component:

* Easy to handle
* Get rid of copying the same logic in every component
* Makes code more readable

**\*\* HOOKS**  
  
**What is react-redux?**

### [**React-redux**](https://www.geeksforgeeks.org/introduction-to-react-redux) is a state management tool which makes it easier to pass these states from one component to another irrespective of their position in the component tree and hence prevents the complexity of the application. As the number of components in our application increases it becomes difficult to pass state as props to multiple components. To overcome this situation, we use react-redux

There are four fundamental concepts of redux in react which decide how the data will flow through components

* Redux Store: It is an object that holds the application state
* Action Creators: These are functions that return actions (objects).
* Actions: Actions are simple objects which conventionally have two properties- type and payload
* Reducers: Reducers are pure functions that update the state of the application in response to actions

**Benefits of using redux**

They are several benefits of using react-redux such as:

* It provides centralized state management i.e. a single store for whole application
* It optimizes performance as it prevents re-rendering of component
* Makes the process of debugging easier
* Since it offers persistent state management therefore storing data for long times become easier

**\*\* HOISTING**

**1. What is hoisting in JavaScript?**  
Hoisting is a behavior in JavaScript where variable and function declarations are moved to the top of their containing scope during the compile phase, before the code is executed. However, only the declarations are hoisted, not the initializations

***---------------------------------------------------EXAMPLE CODES---------------------------------------------------------***  
***-----------------------------------------------------------------------------------------------------------------------------------***

### **2. Does hoisting work with let and const?**

### Yes, let and const declarations are hoisted, but they are not initialized. This leads to a "temporal dead zone" (TDZ) where accessing these variables before their declaration throws a ReferenceError.

Example:

console.log(x); // ReferenceError: Cannot access 'x' before initialization  
let x = 10;

Explanation:

x is hoisted to the top, but since it’s declared with let, it’s in the TDZ until the declaration is encountered. Accessing it before the declaration causes a ReferenceError.

**3. What happens if you try to call a function before it is declared in JavaScript?**

If the function is declared using a **function declaration**, you can call it before the declaration because the function declaration is hoisted. However, if the function is assigned to a variable (i.e., using a function expression), you cannot call it before the declaration because only the variable declaration is hoisted, not the function itself.

Example (function declaration):

greet(); // "Hello!"  
function greet() {

console.log('Hello!');  
}

Example (function expression):

Javascript

Copy code

greet(); // TypeError: greet is not a function  
  
var greet = function() {  
 console.log('Hello!');  
};

Explanation:

In the first case, the function declaration is hoisted, so calling it before the declaration works. In the second case, the function expression is hoisted as a var declaration, so the variable is undefined until the assignment happens, resulting in a TypeError.

### **4. What is the difference between function declarations and function expressions in terms of hoisting?**

**Answer:**

* **Function declarations** are hoisted with both their **name** and **body**. This means you can call them before they appear in the code.
* **Function expressions** (whether using var, let, or const) only hoist the variable declaration, not the function definition. Hence, trying to call a function expression before the assignment results in a TypeError (if assigned to a var) or a ReferenceError (if assigned to let or const).

Example (function declaration vs function expression):

javascript

Copy code

// Function Declaration  
sayHello(); // Works because the declaration is hoisted  
  
function sayHello() {  
 console.log("Hello");  
}  
  
// Function Expression  
greet(); // TypeError: greet is not a function  
  
var greet = function() {  
 console.log("Hello");  
};

### **5. Explain the behavior of hoisting with var, let, and const.**

**Answer:**

* **var**: The declaration is hoisted, but not the assignment. If you try to use a var-declared variable before the assignment, it will be undefined.
* **let and const**: The declarations are hoisted but are in the "temporal dead zone" (TDZ) until the line of declaration is executed. Accessing them before declaration results in a ReferenceError.

Example:

javascript

Copy code

console.log(x); // undefined (for var) or ReferenceError (for let/const)  
var x = 5;  
  
// let and const example  
console.log(a); // ReferenceError: Cannot access 'a' before initialization  
let a = 10;

Explanation:

* For var x, the declaration is hoisted but not the initialization, so undefined is printed.
* For let a, the declaration is hoisted, but accessing a before it’s initialized leads to a ReferenceError.

### **6. What will be the output of the following code?**console.log(foo()); var foo = function() { return "Hello!"; };

**Answer:**

The output will be undefined, not "Hello!".

Explanation:

In this code, the variable foo is hoisted, but its function assignment happens later. Thus, when foo() is called, it refers to the undefined value, not the function, resulting in a TypeError.

To fix this, declare the function expression after its usage, or use a function declaration instead.

### **7. What is the "temporal dead zone" (TDZ) in JavaScript?**

**Answer:**

The "temporal dead zone" refers to the time between the entering of a scope (e.g., inside a function or block) and the actual declaration of a variable using let or const. During this time, any reference to the variable will result in a ReferenceError, even though the variable is hoisted.

Example:

javascript

Copy code

console.log(x); // ReferenceError: Cannot access 'x' before initialization  
let x = 10;

Explanation:

Even though let x is hoisted, the variable x is not available until the line let x = 10; is executed. Accessing x before that will throw a ReferenceError.

### **8. What will be the output of the following code?**

javascript

Copy code

console.log(a);  
let a = 10;

**Answer:**

ReferenceError: Cannot access 'a' before initialization

Explanation:

let is hoisted but placed into the TDZ, so trying to access a before its declaration results in a ReferenceError.

### **9. Explain the difference between undefined and ReferenceError in the context of hoisting.**

**Answer:**

* **undefined**: This happens when you try to access a variable declared with var before it's initialized. The variable exists, but it's not assigned a value yet.
* **ReferenceError**: This occurs when you try to access a variable declared with let or const before it's initialized, or if you try to access a variable that has not been declared at all.

Example:

javascript

Copy code

// var  
console.log(x); // undefined  
var x = 5;  
  
// let/const  
console.log(y); // ReferenceError: Cannot access 'y' before initialization  
let y = 10;

### **10. Can you explain the following hoisting behavior in this code?**

console.log(num); // undefined  
 var num = 3;

**Answer:**

In this case, the declaration of num (var num) is hoisted to the top, but the initialization (num = 3) happens later in the code. Therefore, when console.log(num) is called before the assignment, it outputs undefined.

**\*\*SUBMODULE ADD\*\***  
<https://devconnected.com/how-to-add-and-update-git-submodules/>

**\*\*SUBMODULE CONFLICT\*\***  
<https://stackoverflow.com/questions/826715/how-do-i-manage-conflicts-with-git-submodules>**\*\*Submodule conflict resolve command\*\***  
git restore --staged submodule\_path

**\*\*Submodule Update Commands\*\***  
git submodule update --remote –merge  
git submodule update --init --recursive –remote  
npm install