<https://backbencher.dev/articles/react-hooks-interview-questions>

**Question:**

What are hooks in React?

**Answer:**

Hooks are a new feature added in React v16.8. It allows to use all React features without writing class components. For example, before version 16.8, we need a class component to manage state of a component. Now we can keep state in a functional component using useState hook.

----o----

**Question:**

Will React hooks work inside class components?

**Answer:**

No

----o----

**Question:**

Why React hooks was introduced?

**Answer:**

One reason to introduce hooks was the complexity in dealing with this keyword inside class components. If not handled properly, this will take some other value. That will result in breaking lines like this.setState() and other event handlers. Using hooks, we avoid that complexity when working with functional components.

Class components do not minify very well and also make hot reloading unreliable. That is another inspiration to bring hooks.

Another reason is that, there is no specific way to reuse stateful component logic. Even though HOC and render props patterns address this problem, that asks for modifying the class component code. Hooks allow to share stateful logic without changing the component hierarchy.

Fourth reason is, in a complex class component, related code are scattered in different lifecycle methods. Example, in case of a data fetching, we do that mainly in componentDidMount() and componentDidUpdate(). Another example is, in case of event listeners, we use componentDidMount() to bind an event and componentWillUnmount() to unbind. Hooks instead helps to place related code together.

----o----

**Question:**

How useState hook works? What is/are the arguments accepted by this hook and what is returned by the hook?

**Answer:**

useState hook is a function which is used to store state value in a functional component. It accepts an argument as the initial value of the state. It returns an array with 2 elements. First element is the current value of state. Second element is a function to update the state.

We import useState first from React by

import React, { useState } from "react";

Later we use useState like:

const [currentStateValue, functionToUpdateState] = useState(initialStateValue);

----o----

**Question:**

Here we have a class component with a state value. Each time the button in component is clicked, the count is incremented.

class Counter extends Component {

state = {

count: 0,

};

incrementCount = () => {

this.setState({

count: this.state.count + 1,

});

};

render() {

return (

<div>

<button onClick={this.incrementCount}>Count: {this.state.count}</button>

</div>

);

}

}

Rewrite this component using React hooks.

**Answer:**

import React, { useState } from "react";

function Counter() {

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

return (

<div>

<button

onClick={() => {

setCount(count + 1);

}}

>

Count: {count}

</button>

</div>

);

}

----o----

**Question:**

Below we have a class component. It contains code to update the state based on previous state value.

class Counter extends Component {

state = {

count: 0,

};

incrementCount = () => {

this.setState((prevState) => {

return {

count: prevState.count + 1,

};

});

};

decrementCount = () => {

this.setState((prevState) => {

return {

count: prevState.count - 1,

};

});

};

render() {

return (

<div>

<strong>Count: {this.state.count}</strong>

<button onClick={this.incrementCount}>Increment</button>

<button onClick={this.decrementCount}>Decrement</button>

</div>

);

}

}

Rewrite the above code using React hooks.

**Answer:**

One can update the value of a state variable just by passing the new value to update function or by passing a callback function. Second technique which accepts a callback function is safe to use.

import React, { useState } from "react";

function Counter() {

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

const incrementCount = () => {

setCount((prevCount) => {

return prevCount + 1;

});

};

const decrementCount = () => {

setCount((prevCount) => {

return prevCount - 1;

});

};

return (

<div>

<strong>Count: {count}</strong>

<button onClick={incrementCount}>Increment</button>

<button onClick={decrementCount}>Decrement</button>

</div>

);

}

----o----

**Question:**

Here we have class component that updates the state using the input from a form.

export class Profile extends Component {

state = {

name: "Backbencher",

age: 23,

};

onNameChange = (e) => {

this.setState({

name: e.target.value,

});

};

onAgeChange = (e) => {

this.setState({

age: e.target.value,

});

};

render() {

return (

<div>

<form>

<input

type="text"

value={this.state.name}

onChange={this.onNameChange}

/>

<input

type="text"

value={this.state.age}

onChange={this.onAgeChange}

/>

<h2>

Name: {this.state.name}, Age: {this.state.age}

</h2>

</form>

</div>

);

}

}

Rewrite the same component using React hooks.

**Answer:**

import React, { useState } from "react";

function Profile() {

const [profile, setProfile] = useState({

name: "Backbencher",

age: 24,

});

const onNameChange = (e) => {

setProfile({ ...profile, name: e.target.value });

};

const onAgeChange = (e) => {

setProfile({ ...profile, age: e.target.value });

};

return (

<div>

<form>

<input type="text" value={profile.name} onChange={onNameChange} />

<input type="text" value={profile.age} onChange={onAgeChange} />

<h2>

Name: {profile.name}, Age: {profile.age}

</h2>

</form>

</div>

);

}

The setter function of useState() does not automatically merge if an object is stored in state. But in case of setState() method in class components, auto merging happens.

Here we are merging object properties with the help of JavaScript spread operator.

----o----

**Question:**

What are the differences in using hooks and class components with respect to state management?

**Answer:**

When using setState() in class components, always the state variable is an object. Where as, the state variable in hooks can be of any type like number, string, boolean, object or array.

When state variable is an object, setState() in class components automatically merges the new value to the state object. But in case of setter function in useState(), we need to explicitly merge the updated object property using spread operator.

----o----

**Question:**

What is the purpose of useEffect hook?

**Answer:**

The Effect hook lets us to perform side effects in functional components. It helps us to avoid redundant code in different lifecycle methods of a class component. It helps to group related code.

----o----

**Question:**

Here is a class component that prints *Boom* in console whenever it is mounted or updated.

export class Banner extends Component {

state = {

count: 0,

};

updateState = () => {

this.setState({

count: this.state.count + 1,

});

};

componentDidMount() {

console.log("Boom");

}

componentDidUpdate() {

console.log("Boom");

}

render() {

return (

<div>

<button onClick={this.updateState}>State: {this.state.count}</button>

</div>

);

}

}

Remove the redundant console.log statement using React hooks.

**Answer:**

componentDidMount() and componentDidUpdate() are lifecycle methods. Such side effects can be done using useEffect hook. useEffect hook is a function which accepts a callback function. That callback function is called every time render happens.

The code can be rewritten as:

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

function Banner() {

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

useEffect(() => {

console.log("Boom");

});

const updateState = () => {

setCount(count + 1);

};

return (

<div>

<button onClick={updateState}>State: {count}</button>

</div>

);

}

----o----

**Question:**

Understand the code below:

function Banner() {

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

const [name, setName] = useState("");

useEffect(() => {

console.log("Count is updated");

});

return (

<div>

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

<input

type="text"

value={name}

onChange={(e) => setName(e.target.value)}

/>

</div>

);

}

It logs "Count is updated" message even when updating the value in textbox. How can we show the log message only when the count state is updated?

**Answer:**

useEffect function accepts a second parameter which should be an array. Within this array, we need to pass the props or state we need to watch for. Only if those props or state mentioned in the array change, the effect is executed. So in our code, we add the second argument and specify only count value in the array.

Here is the udpated useEffect code:

useEffect(() => {

console.log("Count is updated");

}, [count]);

----o----

**Question:**

We have got a class component that updates time every second. It uses componentDidMount() to set the timer.

export class Clock extends Component {

state = {

date: new Date(),

};

componentDidMount() {

setInterval(() => {

this.setState({

date: new Date(),

});

}, 1000);

}

render() {

return <div>{this.state.date.toString()}</div>;

}

}

Convert the above code to React hooks.

**Answer:**

componentDidMount() is a lifecycle method that executes only once in a component lifecycle. We use useEffect to bring effects of componentDidMount(). But useEffect runs on every props or state updation. To prevent it, we make use of second array argument of useState. We keep that array empty. So for React, there are no props or state to watch for. Therefore useEffect runs only once like componentDidMount().

Here is the code using React hooks.

function Clock() {

const [date, setDate] = useState(new Date());

useEffect(() => {

setInterval(() => {

setDate(new Date());

}, 1000);

}, []);

return <div>{date.toString()}</div>;

}

----o----

**Question:**

We have a code snippet from a class component which registers and remove an event listener.

componentDidMount() {

window.addEventListener("mousemove", this.handleMousePosition);

}

componentWillUnmount() {

window.removeEventListener("mousemove", this.handleMousePosition);

}

Convert this code to React hooks format.

**Answer:**

useEffect(() => {

window.addEventListener("mousemove", handleMousePosition);

return () => {

window.removeEventListener("mousemove", handleMousePosition);

};

}, []);

----o----

**Question:**

Here is a code snippet from a Context consumer component.

import { NameContext, AgeContext } from "./ProviderComponent";

export class ConsumerComponent extends Component {

render() {

return (

<NameContext.Consumer>

{(name) => {

return (

<AgeContext.Consumer>

{(age) => (

<div>

Name: {name}, Age: {age}

</div>

)}

</AgeContext.Consumer>

);

}}

</NameContext.Consumer>

);

}

}

Rewrite the ConsumerComponent using useContext React hook.

**Answer:**

Hooks can be used only in a functional component. The ConsumerComponent can be re-written as:

function ConsumerComponent() {

const name = useContext(NameContext);

const age = useContext(AgeContext);

return (

<div>

Name: {name}, Age: {age}

</div>

);

}

<https://michael-pautov.medium.com/top-10-interview-questions-related-to-react-hooks-dfdada3c0040>

**Q1. What Are React Hooks?**

These are in-built functions that allow developers to use state and lifecycle methods within components in React. Each component’s lifecycle has 3 phases which are mount, unmount, and update. Alongside that, components have states and properties. Hooks allow developers to use these methods whilst improving code reuse with greater flexibility navigating the component tree.

**Q2. What is the primary benefit of Hooks?**

Through Hooks, you can code in React without using classes. With Hooks, you can reuse existing states in your code plus easily test them later on for errors and bugs in the code. Through the use of Hooks, you can tightly couple logic in the code on React.

**Q3. What Are The 2 Rules You Must Follow While Using Hooks?**

There are 2 rules which are imposed while you are coding with Hooks:

1. React Hooks should only be called at the Top Level. They shouldn’t be called inside loops, nested functions or conditions.
2. Hooks can only be called from React Function Components.

**Q4. What is the difference between setState() and useEffect() in React?**

Simply put, setState() allows programmers to set the state of functional components whereas the useEffect() method allows programmers to use lifecycle methods to React.

**Q5. Explain State Hook with an example**

State Hooks are primarily used to refer to the states of the components in React and can be used to change them on the runtime as per the functionality of your web application. An example below increments a counter on click of a button:

function Counter() {  
  
 const [count, setCount] = useState(0)  
  
 return (  
 <div>  
 <p>You clicked {count} times</p>  
 <button onClick={() => setCount(count + 1)}>  
 Increment Counter  
 </button>  
 </div>  
 )  
}

In the above example, we declare a constant count whose state initially is 0. Depending on how many times the button on the webpage is clicked, the count variable will be incremented that many times.

**Q6. Is it possible to use both classes and Hooks in my React Code?**

While Hooks cannot be a part of classes, you can use both within your React Web applications as you wish. Existing React applications will continue to work fine and there are no plans of removing classes from React in the near future. However, Hooks offer greater flexibility and easier code understanding therefore your company or organization should quickly transition to it.

**Q7. Do Hooks work with static typing?**

Static typing is a process of checking code during compile time to ensure that all variables are statically typed. Hooks are primarily functions and have been designed in a way to ensure that all attributes are statically typed. You can also use the React API with custom Hooks if you want to enforce stricter static typing within your code.

**Q8. What functionalities that are provided by classes and are not covered by Hooks yet?**

The following methods have not been introduced in Hooks yet:

1. getSnapshotBeforeUpdate
2. getDerivedStateFromError
3. componentDidCatch

These are uncommon lifecycles, however. Furthermore, Hooks may not be compatible with certain third-party libraries.

**Q9. How do you use multiple states with Hooks?**

The example below illustrates the use of multiple states with React Hooks:

function MultipleStates() {  
 // Declare multiple state variables!  
 const [car\_model, setModel] = useState(1998);  
 const [car\_name, setName] = useState('Toyota Corolla');  
 const [car\_price, setPrice] = useState(600000);  
 // ...  
}

As you can see, we have declared multiple states within a single function. We have the car price, model, and name in a single function MultipleStates() above.

**Q10. Explain Effect Hook with an example**

If an action is performed on a webpage, an Effect Hook may be used to transfer a side effect to another part of the DOM. The example below updates the document title of the webpage when a button is clicked:

function Effect() {  
 const [car, setcar] = useState('Ferrari');  
 useEffect(() => {  
  
 ***document***.title = `My favorite car is ${car}`;  
 });  
  
 return (  
 <div>  
 <p>My favorite car is {car}</p>  
 <button onClick={() => setcar('McLaren')}>  
 Not this one?  
 </button>  
 </div>  
 );  
}

The above example changes the favorite car of the user from Ferrari to McLaren upon the click of a button. The effect of the change is also transferred using useEffect()to the document title and it is also updated simultaneously.

In essence, useEffect() is replacing componentDidMount and componentDidUpdate in this code.

<https://www.dotnettricks.com/learn/react/react-hooks-full-guide>

## What are the benefits of using Hooks?

For React developers, Hooks convey ample benefits as follows:

* It revolutionizes the way you write components
* You can write concise and clearer code.
* Hooks are simpler to work with and test. Code would appear cleaner and easier to read.
* A related logic could be tightly coupled inside a custom hook**.**
* It simplifies how to make code more composable and reusable.
* Unlike HOCs, they don’t create another element in DOM.
* Hooks would work more efficiently with the future React optimizations. For example, it would work ahead of time compilation as well as components folding.
* Components folding may be implemented in the future. It suggests that for code elimination at compile-time, hooks are simple to reuse the stateful logic.

### Rules for using Hooks:

Before creating Hooks in React, you need to follow some rules, else you will face troubles later on. These rules make sure all the stateful logic within a component is visible inside its source code. Though these rules may not be mentioned in the [**React tutorial**](https://www.dotnettricks.com/learn/react), it is better to get familiar with them beforehand. Here are the Hook rules:

**1#. Call Hooks at the top level only:**

Make sure not to call Hooks within loops, nested functions, or conditions. It is important to note that Hooks must always be utilized at the superior level of the React functions. It suggests that Hooks are invoked in the same order every time a component is rendered.

**2#. Call Hooks from React functions only:**

It is not allowed to call Hooks from standard JavaScript functions. The alternative is you can call Hooks from [React function components](https://www.dotnettricks.com/learn/react/functional-component-react). It is allowed to call Hooks from custom Hooks.

**3#. When can you use Hooks?**

When you write a function component, you would wish to add some state to it. Formerly, this was possible by converting it to a class. Now you can do that with the help of Hooks within the existing function component. After you know the appropriate usage of Hooks, it becomes easy to understand the Hooks concept in the [**react learning path**](https://www.dotnettricks.com/paths/become-react-developer).

## Hooks Effect:

The Effect Hook enables you to accomplish side effects i.e. an action within the function components. Those components' lifecycle methods available within class components are not used in Hooks Effect. In simple terms, Effects Hooks are correspondent to componentDidUpdate(), componentDidMount(),and componentWillUnmount() lifecycle methods. If you are preparing for [**React Interview Questions and Answers**](https://www.dotnettricks.com/learn/react/top-20-react-interview-questions-answers), you can expect questions on Hooks Effect.

The corresponding side effects have some common features that the majority of web applications should perform. They are:

* Fetching as well as consuming data from a server API
* Updating the DOM
* Setting up a subscription, and more

Two types of side effects in React component are:

1. Effects Without Cleanup
2. Effects With Cleanup

**i. Effects without Cleanup:**

This side effect is used in useEffect that does not restrict the browser from updating the screen. Moreover, it enhances the responsiveness of the app. Some common examples of the effects without cleanup are network requests, manual DOM mutations, Logging, etc.

**ii. Effects with Cleanup:**

Certain Hooks effects need cleanup after updating the DOM. For instance, if you intend to set up a subscription to an external data source, you must clear up memory else there will be a memory leak problem. It is found that React carries out the cleanup of memory whenever the component is unmounted. But the effects run for every render method instead of any particular method. Thus, React would also clean up effects from the preceding render before executing the effects succeeding time.

## Types of Hooks:

The most common types of Hooks are built-in Hooks and Custom Hooks.

**Built-in Hooks:**

The built-in Hooks are categorized into 2 parts as follows:

**Basic Hooks:**

* useState
* useContext
* useEffect

**Additional Hooks:**

* useReducer
* useMemo
* useCallback
* useRef
* useImperativeHandle
* useDebugValue
* useLayoutEffect

**Custom Hooks:**

A custom Hook is basically a JavaScript function. Its name begins with "use" that could invoke other Hooks. The working of a custom Hook is identical to a regular function. The "use" in the starting describes that this particular function conforms to the rules of Hooks. Moreover, building custom Hooks enables you to extract component logic inside reusable functions.

Before proceeding with the installation of Hooks, you need to know its pre-requisites as follows:

**Pre-requisites for React Hooks:**

* Node version 6 or higher
* NPM version 5.2 or higher
* Create-react-app tool for operating the React App

There are 10 Hooks in total 🔥

### 🚀 useState :

It is the most important and often used hook. The purpose of this hook to handle reactive data, any data that changes in the application is called state, when any of the data changes, React re-renders the UI.

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

### 🚀 useEffect :

It allows us to implement all of the lifecycle hooks from within a single function API.

// this will run when the component mounts and anytime the stateful data changes

React.useEffect(() => {

alert('Hey, Nads here!');

});

// this will run, when the component is first initialized

React.useEffect(() => {

alert('Hey, Nads here!');

}, []);

// this will run only when count state changes

React.useEffect(() => {

fetch('nads').then(() => setLoaded(true));

}, [count]);

// this will run when the component is destroyed or before the component is removed from UI.

React.useEffect(() => {

alert('Hey, Nads here');

return () => alert('Goodbye Component');

});

### 🚀 useContext :

This hook allows us to work with React's Context API, which itself a mechanism to allow us to share data within it's component tree without passing through props. It basically removes prop-drilling

const ans = {

right: '✅',

wrong: '❌'

}

const AnsContext = createContext(ans);

function Exam(props) {

return (

// Any child component inside this component can access the value which is sent.

<AnsContext.Provider value={ans.right}>

<RightAns />

</AnsContext.Provider>

)

}

function RightAns() {

// it consumes value from the nearest parent provider.

const ans = React.useContext(AnsContext);

return <p>{ans}</p>

// previously we were required to wrap up inside the AnsContext.Consumer

// but this useContext hook, get rids that.

}

### 🚀 useRef :

This hook allows us to create a mutable object. It is used, when the value keeps changes like in the case of useState hook, but the difference is, it doesn't trigger a re-render when the value changes.

The common use case of this, is to grab HTML elements from the DOM.

function App() {

const myBtn = React.useRef(null);

const handleBtn = () => myBtn.current.click();

return (

<button ref={myBtn} onChange={handleBtn} >

</button>

)

}

### 🚀 useReducer :

It does very similiar to setState, It's a different way to manage state using Redux Pattern. Instead of updating the state directly, we dispatch actions, that go to a reducer function, and this function figure out, how to compute the next state.

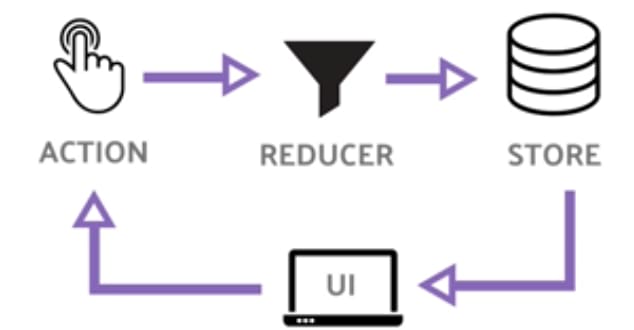
[](https://res.cloudinary.com/practicaldev/image/fetch/s--ik_MbBeK--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/885ulztfmsm2xhdqj3mp.png)

Fig. useReducer Architecture

function reducer(state, dispatch) {

switch(action.type) {

case 'increment':

return state+1;

case 'decrement':

return state-1;

default:

throw new Error();

}

}

function useReducer() {

// state is the state we want to show in the UI.

const [state, dispatch] = React.useReducer(reducer, 0);

return (

<>

Count : {state}

<button onClick={() => dispatch({type:'decrement'})}>-</button>

<button onClick={() => dispatch({type:'increment'})}>+</button>

</>

)

}

### 🚀 useMemo :

This hook will help you to optimise computational cost or improve performance. It mostly used when we're needed to make expensive calculations.

function useMemo() {

const [count, setCount] = React.useState(60);

const expensiveCount = useMemo(() => {

return count\*\*2;

}, [count]) // recompute when count changes.

}

Works great for memoizing returned values, but in other CSSNamespaceRule, we want to memoize the whole function, in that case we can use this hook ↓

### 🚀 useCallback :

function useCallbackDemo() {

const [count, setCount] = useState(60);

const showCount = React.useCallback(() => {

alert(`Count ${count}`);

}, [count])

return <> <SomeChild handler = {showCount} /> </>

}

### 🚀 useImperativeHandle :

This hook is use to modify the exposed ref and it is rarely used.

function useImperativeHandleDemo(props, ref) {

const myBtn = useRef(null);

React.useImperativeHandle(ref, () => ({

click: () => {

console.log('clicking button!');

myBtn.current.click();

}

}));

}

### 🚀 useLayoutEffect :

It works same as useEffect hook with one difference, the callback will run after rendering the component but before the actual updates have been painted to the screen.

⚠️ : Blocks visual updates until your callback is finished.

function useLayoutEffectDemo() {

const myBtn = React.useRef(null);

React.useLayoutEffect(() => {

const rect = myBtn.current.getBoundingClientRect();

// scroll position before the dom is visually updated

console.log(rect.height);

})

}

### 🚀 useDebugValue :

This hook doesn't make much sense, but it allows us to define our own custom labels in React Dev Tools, which are useful for debugging. Suppose we have n number of components which uses the same logic, then we can separately define our own function and that can be used in other components, but the key thing here is we can debug things

function useDisplayName() {

const [displayName, setDisplayName] = React.useState();

React.useEffect(() => {

const data = fetchFromDatabase(props.userId);

setDisplayName(data.displayName);

}, []);

React.useDebugValue(displayName ?? 'loading...');

return displayName;

}

The return value can be used in other components or else where in the application like this 👇🏽

function App() {

const displayName = useDisplayName();

return <button>{displayName}</button>;

}