



# React interview questions



ALEX BOOKER

19 NOV 2020 • 17 MIN READ

If you want to land a great React job in 2020 or 2021, this is the post for you 😊.

We're bringing you this post on the back of our new [React Interview Questions](#) course with the awesome [@Cassidoo](#) 🎉

In that course, Cassidoo draws on her professional experience working at Netlify (and before that, CodePen) to share 28 likely React interview questions and example answers.

You're reading an epic 4500 word version of those same common React interview questions and example answers. Use this as a quick reference or as an exercise to rehearse your answers aloud. We have also included a PDF below in case you'd like something to download and print 😎.


Here, we're listing all the same questions plus vetted answers for you to adapt. Use this as inspiration to phrase eloquent and confident answers that will WOW 🏆 your soon-to-be employer.

For each question, we aim to highlight:

- 🗝️ The key thing to mention in your answer



## React interview questions

-  In some cases, we'll also mention common wrong answers for you to avoid at all costs

Without further ado, here are the questions (listed in the same order that they appear in the course, in case you'd like to use these resources together):

SUBJECT	QUESTION
React DOM	<u>What is the difference between the virtual DOM and the</u>
	<u>Is the virtual DOM the same as the shadow DOM?</u>
React limitations	<u>What are the limitations of React?</u>
JSX	<u>What is JSX?</u>
	<u>What is the difference between an element and compon</u>
	<u>Can you write React without JSX?</u>
Props	<u>How do you pass a value from a parent to child?</u>
	<u>How do you pass a value from child to parent?</u>
	<u>What is prop drilling?</u>
	<u>Can a child component modify its own props?</u>
State and lifecycle	<u>What is the difference between props and state?</u>
	<u>How does state in a class component differ from state in</u>
	<u>What is the component lifecycle?</u>
	<u>How do you update lifecycle in functional components?</u>
Effects	<u>What argument does <code>useEffect</code> take?</u>
	<u>When does the <code>useEffect</code> function run?</u>
	<u>What is the <code>useEffect</code> function's return value?</u>
Refs	<u>What is the difference between ref and state variables?</u>



## React interview questions

	<u>When is the best time to use refs?</u>
	<u>What is the proper way to update a ref in a function component?</u>
Context	<u>What is the difference between the Context API and props?</u>
	<u>When shouldn't you use the Context API?</u>
Other	<u>What is a Fragment?</u>
	<u>When should you create class-based component versus functional component?</u>
	<u>What is a higher order component?</u>
	<u>What is portal?</u>
	<u>What are uncontrolled and controlled components?</u>

[Click here to download the PDF](#) 

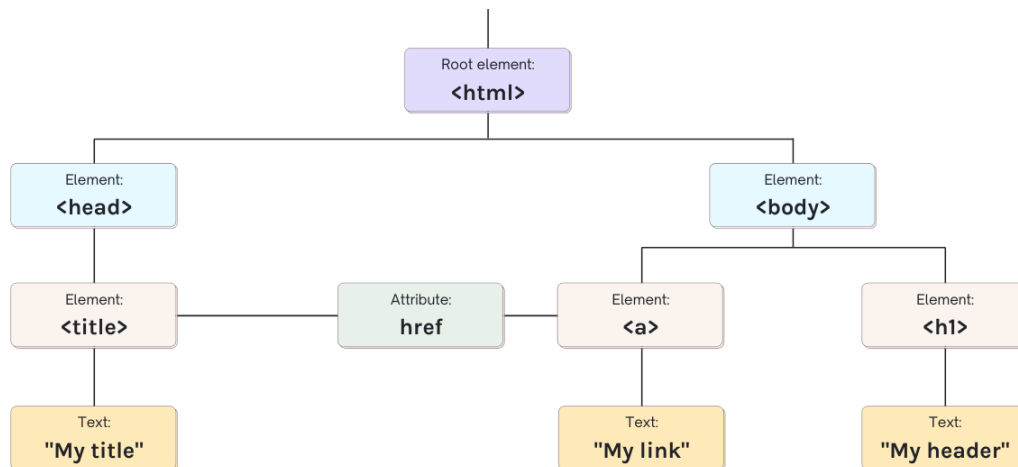
## React DOM

### What is the difference between the virtual DOM and the real DOM?

The DOM represents an HTML document as a tree structure wherein each node represents part of the document (for example, an element, element attribute, or text):



## React interview questions



Using vanilla JavaScript and the DOM API, you can access any element you like (for example, using `document.getElementById`) and update it directly.

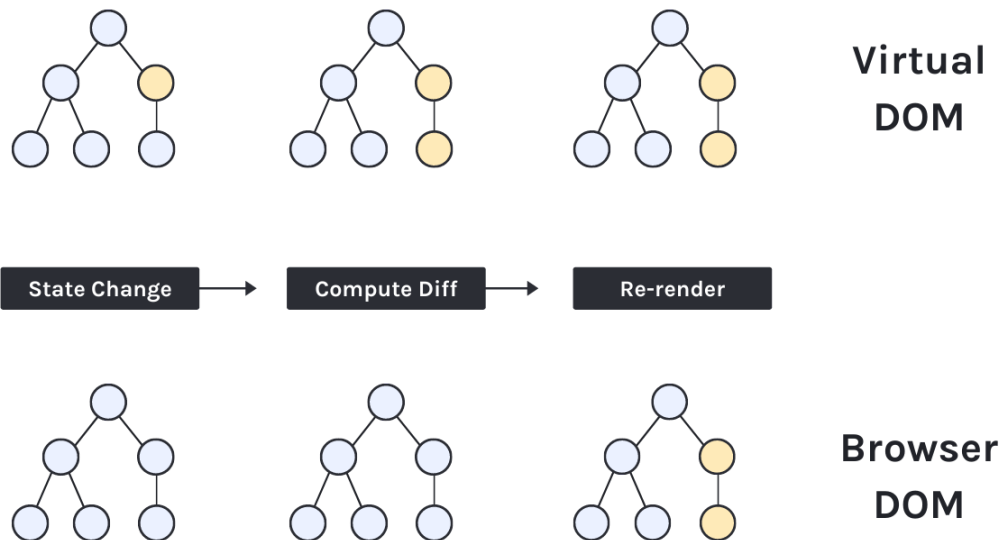
When you do this, the browser engine traverses the DOM and re-renders each node *even if that node hasn't changed since the previous render*. This can be noticeably inefficient 🤔

Imagine a scenario where you need to update only one `tr` of 10,000 in a `table`. Rendering all 10,000 rows will almost certainly lead to a drop in frames, potentially causing the table to flicker and interrupt the user's experience.

This is where React's virtual DOM (VDOM) comes into play ✅.

React increases your UI's performance by building a "virtual" representation of the DOM (a VDOM 😊) to keep track of all the changes it needs to make to the real DOM.

Every time your app's state updates, React builds a new VDOM and *diffs* with the old one to determine what changes are necessary before updating the DOM directly and efficiently:



- 🗝️ The important thing to mention here is *diffing*. If you want to flex a little, you can describe this process by its technical name, which is *reconciliation* (React *reconciles* the newly-built VDOM with the previous one)
- 📖 Learn more
  - [React's documentation on VDOM](#)
  - For an alternative viewpoint, we also recommend you read [Virtual DOM is pure overhead](#)
- 🚫 A common misconception is that the VDOM is a React feature. This is not true! VDOM is a programming concept that predates React and is adopted by many UI libraries, including Vue



## Is the virtual DOM the same as the shadow DOM?



## React interview questions

Whereas the virtual DOM is a programming concept implemented by React predominantly to increase rendering performance, the Shadow DOM is a browser technology designed for scoping variables and CSS in web components.

The virtual DOM and Shadow DOM sound similar in name, but that is where the similarity begins and ends - they are totally unrelated.

-  You shouldn't need to know the ins and outs of Shadow DOM to succeed in React technical interview
-  Learn more
  - [This screencast](#) provides a short and sweet explanation of Shadow DOM that will make sense even if you don't know much about web components

## React limitations

### What are the limitations of React?

No tool is without its limitations, and React is no exception.

Weighing in at 133kb, React is considered to be a *relatively* heavy dependency. By comparison, Vue is 58kb. For this reason, React could be considered overkill for small apps.

Comparing React and Vue in file size feels fair because they're both libraries as opposed to frameworks.

Compared to a framework like Angular, React doesn't enforce strong opinions about how to write and structure your code or about which libraries to use for things like data fetching - with Angular, team



## React interview questions

libraries like Axios or Fetch.

Because React does not enforce opinions about how to best structure code, teams need to be especially diligent about writing code consistently so that the project can evolve deliberately. This can lead to communication overhead and steepen the learning curve for newbies.

These are important considerations to make when embarking on a new project. Once you commit to React, one limitation is that the documentation is not always linear or up to date 😊.

- 🔑 Show the interviewer you can think critically about which tool you apply to which problems rather than blindly reaching for React
- 📖 Learn more
  - As a bonus, you can learn about the limitations of React Native as many teams find the "write once run everywhere" idea alluring until they try it

## JSX

### What is JSX?

Similar in appearance to XML and HTML, JavaScript XML (JSX) is used to create elements using a familiar syntax.

JSX is an extension to JavaScript understood only by preprocessors like Babel. Once encountered by a preprocessor, this HTML-like text is converted into regular old function calls to `React.createElement` :

A diagram illustrating the relationship between JSX and the React.createElement function. At the top, a dark horizontal bar contains the text 'MyButton' in a light blue font. Below this bar, a large, light blue, semi-transparent circle is centered. A black arrow points downwards from the circle towards a dark rectangular box. Inside this box, the code for React.createElement is written in a light blue font: 

```
React.createElement(  
  MyButton,  
  {color: 'blue', shadowSize: 2},  
  'Click Me'  
)
```

- 🗝️ JSX is syntactic sugar for the `React.createElement` function
- 📖 Learn more
  - The React documentation has an [introduction to JSX](#) and a [deep dive on JSX](#) - while we cannot promise these two resources alone will make you an expert, they do cover almost everything there is to know
- 🚫 While JSX is HTML-like, it is not HTML. If you're tempted to answer, "JSX allows you to write HTML in your JavaScript", that would not be accurate

## What is the difference between an element and a component?

Coming soon.

## Can you write React without JSX?

In a word, yes.

JSX is not part of the [ECMAScript specification](#), and therefore no web browser actually understands JSX.

Rather, JSX is an *extension* to the JavaScript language only understood



## JSX

### React interview questions

When a preprocessor encounters some JSX code, it converts the HTML-like syntax into regular old function calls to `React.createElement` :

```
<MyButton color="blue" shadowSize={2}>
  Click Me
</MyButton>
```



```
React.createElement(
  MyButton,
  {color: 'blue', shadowSize: 2},
  'Click Me'
)
```

`React.createElement` is part of React's public top-level API just like `React.component` or `React.useRef` (to name a couple). Nothing is stopping you from invoking `React.createElement` in your own code should you choose.

- 🗝️ JSX is syntactic sugar for the `React.createElement` function meaning you *could* call `React.createElement` directly
- 📖 Learn more
  - The answers on this StackOverflow thread reveal all you need to know about the magic, which is JSX, Babel and Webpack ✨

## Props

### How do you pass a value from parent to child?



## React interview questions

- Typically there is an `onChange` prop you need to set. 🏆
- 📖 Learn more:
  - React documentation on [Components and Props](#)

## How do you pass a value from child to parent?

To pass a value from a child component to a parent component, the parent must supply a function for the child component to call with the value.

A common example is a custom form component. Imagine a component to select a language called `SelectLanguage`.

When the language is selected, we'd like to pass that value back *UP* to the parent for processing.

To do this, the `SelectLanguage` component would need to accept a callback function called something like `onLanguageSelect`, which it will then call with the new value.

- 🔑 Pass a function prop to the child, which the child can call. Try and include a common example like a `CustomForm` or `SelectLanguage` form component in your answer props
- 📖 Learn more:
  - We deliberately borrowed the `SelectLanguage` example component from [this StackOverflow answer](#) so you can read more

## What is prop drilling?

Prop drilling is where you pass props from some `FirstComponent` to



## React interview questions

Prop drilling is sometimes called *threading* and is considered to be a slippery slope if not an anti-pattern 🤖.

Imagine drilling a prop 5, 10, maybe more (!) levels deep - that code would quickly become difficult to understand. The trap happens when you need to share data across many different components - data like locale preference, theme preference, or user data.

While prop drilling is not inherently bad, there are normally more eloquent and maintainable solutions to explore like component composition or React Context however, these solutions are not without their limitations.

- 🔑 Prop drilling is what happens when you pass a prop more than two components deep and the second component doesn't actually need the data (it just passes it along)
- 📖 Learn more
  - Kent C. Dodds provides a balanced view on what prop drilling is, why it's bad, and how to avoid common problems with it
  - Although this post by LogRocket is orientated at TypeScript developers (where the downsides of prop drilling are exasperated due to needing additional type definitions), we found it to be an interesting read

## Can a child component modify its own props?

Nu-huh.

A component can update its own state but cannot update its own props.





## React interview questions

value it does not own. Props are, therefore, read-only.

Attempting to modify props will either cause obvious problems or, worse, put your React app in a subtly unstable state.

React dictates that to update the UI, update state.

-  React needs you to treat props as read-only (even if there are ways of messing with them)
-  Learn more
  - This [StackOverflow answer](#) uses example code to illustrate what can happen if you mess with props from a child component
  - While a child cannot update its own props, the value of those props can change if the parent changes them

through a state change. Despite the sensational (possibly confusing) title, [This FreeCodeCamp post](#) shows a familiar example of what this pattern looks like

## State and lifecycle

### What is the difference between props and state?

Props are essentially *options* you initialize a child component with. These options (if you like) belong to the parent component and must not be updated by the child component receiving them.

State, on the other hand, belongs to and is managed by the component.

State is always initiated with a default value, and that value can change over the lifetime of the component in response to events like user input or network responses. When state changes, the component



## React interview questions

State is optional, meaning some components have props but no state. Such components are known as *stateless components*.

### How does state in a class component differ from state in a functional component?

State in a class component belongs to the class instance ( `this` ), whereas state in a functional component is preserved by React between renders and recalled each time.

In a class component, the initial state is set within the component's constructor function then accessed or set using `this.state` and `this.setState()` respectively.

In a functional component, state is managed using the `useState` Hook. `useState` accepts an argument for its initial state before returning the current state and a function that updates the state as a pair.

- 🗝️ State in a class component belongs to the class instance (`this`) and is initialized along with the class in the constructor function. In a functional component, the `useState` Hook is recalled each time the component renders and returns the state remembered by React under the hood
- 📖 Learn more
  - We really enjoyed this post on [Functional Components vs. Class Components in React by Twilio](#) - the section on Handling State is particularly pertinent here

### What is the component lifecycle?



## React interview questions

- 🌱 First, the component is **initialized** and **mounted** on the DOM
- 🌳 Over time the component is **updated**
- 🍂 Eventually, the component is **unmounted** or removed from the DOM

Using lifecycle methods in a class component or the useEffect Hook in a functional component, we can run code at particular times in a component's life.

For example, in a class component, we might implement `componentDidMount` and write code to set-up a new web socket connection. As real-time web socket data trickles in, state is updated, and, in turn, the `render` lifecycle method is run to update the UI. When the component is

no longer needed, we close the web socket connection by

implementing `componentWillUnmount`.



- 🗝️ React components have several lifecycle methods that you can override to run code at particular times in the component's life. Knowing all the functions isn't a bad idea, but it's more important that you can explain when you'd use each. Some lifecycle methods are pretty uncommon, so you're unlikely to have experience with them. Don't lead the interviewer down this path if you don't need to.
- 📖 Learn more
  - This page on React.Component has more details than you will ever need

## How do you update lifecycle in function components?

Using the `useEffect` Hook!



## React interview questions

-  Use `useEffect`
-  Learn more
  - [Using the Effect Hook](#)

## Effects


### What arguments does `useEffect` take?


`useEffect` takes two arguments.

The first argument is a function called `effect` and is what gives the `useEffect` Hook its name.

The second argument is an optional array called `dependencies` and allows you to control when exactly the `effect` function is run. Think of a `dependencies` as variables (typically state variables) that the `effect` function references and therefore depends on.

If you choose not to specify any `dependencies`, React will default to running the `effect` when the component is first mounted and after every completed render. In most cases, this is unnecessary, and it would be better to run the `effect` only if something has changed.


This is where the optional `dependencies` argument comes in .

When `dependencies` is present, React compares the current value of a `dependencies` with the values used in the previous render. `effect` is only run if `dependencies` has changed .

If you want `effect` to run only once (similar to `componentDidMount`), you can pass an empty array ( `[]` ) to `dependencies`.



## React interview questions



-  Learn more
  - What is useEffect hook and how do you use it?

## When does the useEffect function run?

When `useEffect` runs exactly depends on the `dependencies` array argument:

- If you pass an empty array (`[]`), the effect runs when the component is mounted (similar to `componentDidMount`)
- If you pass an array of state variables (`[var]`), the effect runs when the component is mounted, and anytime the values of these variables change
- If you omit the dependencies argument, the effect is run when the component is mounted and on each state change

That is about the sum of it!

-  That is about the sum of it!
-  Learn more
  - Hooks API reference

## What is the useEffect function's return value?

The `useEffect` function takes two arguments - an `effect` function and an optional `dependencies` array.

The `effect` function returns either nothing (`undefined`) or a function we can call `cleanup`.







## React interview questions

`componentWillUnmount` ).

Additionally, if a component renders multiple times (as they typically do), the previous effect is cleaned up before executing the next effect.

-  Returns a cleanup function (similar to `componentWillUnmount` ) and runs after each effect
-  Learn more
  - [Hooks API reference](#)
  - [Replace lifecycle with hooks in React](#)

## Refs

### What is the difference between refs and state variables?

Both refs and state variables provide a way to persist values between renders; however, only state variables trigger a re-render.

While refs were traditionally (and still are) used to access DOM elements directly (for example, when integrating with a third-party DOM library), it has become increasingly common to use refs in functional components to persist values between renders that should not trigger a re-render when the value is updated.

There isn't much reason to use refs for this reason in class components because it's more natural to store these values in fields that belong to the class instance and would be persisted between renders regardless.

-  Both persist values between renders, but only state



## React interview questions

- Understanding React's useRef Hook

### When is the best time to use refs?



Only use refs when necessary!

Refs are mostly used in one of two ways.

One use of refs is to access a DOM element directly to manipulate it - an example would be when implementing a third-party DOM library. Another example might be to trigger imperative animations.

The second use of refs is in functional components, where they are sometimes a good choice of utility to persist values between renders without triggering the component to re-render if the value changes.

When someone is new to React, refs often feel familiar to them because they are used to freely writing imperative code. For this reason, beginners tend to overreach for refs. We know better. We know that to get the most from React, we must think in React and ideally control every piece of our app with state and component hierarchy. The React documentation describes refs as an "escape hatch" for a good reason!

-  Only use refs when necessary to avoid breaking encapsulation
-  Learn more
  - Why you should use refs sparingly in production

### What is the proper way to update a ref in a function component?



## React interview questions

- 🗝️ That is about the sum of it!
- 📖 Learn more
  - [Hooks API Reference](#)

## Context

### What is the difference between the context API and prop drilling?

In React, you explicitly pass data from parent component to child components through props.

If the child component that needs the data happens to be deeply nested, we sometimes resort to prop-drilling, which can be a slippery slope. This is often the case when data is shared across many different components - data like locale preference, theme preference, or user data (like the authentication state).

Conversely, the Context API affords us a central data store, which we can *implicitly* access to consume data from any component without needing to request it as a prop explicitly.

The implicit nature of the Context API allows for terser code that is easier to manage but can also lead to "gotchas!" if the value is updated unexpectedly as it won't be so easy to trace the value and learn where it was modified linearly.

- 🗝️ Prop-drilling is explicit and therefore long-winded, but at least you know what you're going to get. Context API is implicit and therefore terse but can cause unnecessary re-renders if





## React interview questions

- [Hooks API Reference](#)

### When shouldn't you use the context API?

The Context API's main downside is that every time the context changes, all components consuming the value re-render. This may have negative performance consequences.

For this reason, you should only use Context for infrequently updated data like a theme preference.



-  That is about the sum of it!
-  Learn more
  - [The Context API's dirty little secret](#)

### Miscellaneous (but important!) questions

#### What is Fragment?

`Fragment` is a newly-introduced component that supports returning multiple children from a component's render method without needing an extraneous `div` element.

You can either reference it using React's top-level API (`React.Fragment`) or using JSX syntactic sugar (`<>`).

-  Instead of returning a `div` from a component's render method, we can instead return a `Fragment`
-  Learn more
  - If you need to answer "why fragments?", [this dev.to post](#) is




## React interview questions

and the JSX syntactic sugar

### When should you create class-based component versus a function component?



In the world of React, there are two ways of creating React components. One is to use a class that derives from `React.Component` and the other is to use a functional component with Hooks.

Before Hooks' advent in 2018, it wasn't possible to substitute class-based components with functional components - mainly because there was no way to set state and remember values between renders without writing a class.

With Hooks, classes and functional components are generally interchangeable, and as we enter the new year, the trend is clear: functional components are on the rise and for good reasons 

Functional components unlock all the advantages of hooks, including ease of use, testability, and cleaner code.

At the time of this writing, there are no Hook equivalents to the (uncommon) `getSnapshotBeforeUpdate`, `getDerivedStateFromError`, and `componentDidCatch` lifecycle methods, but they are coming "soon."

-  Class components and functional components are mostly interchangeable. Choose whichever the codebase is already using for consistency. For new projects, use Hooks unless you need a lifecycle method Hooks don't yet support.
-  Learn more
  - [Hooks API Reference](#)





## React interview questions

A higher-order component (HOC) is a function that takes a component and returns a new, modified component.

While HOCs are associated with React, they aren't a React feature so much as a pattern inspired by a functional programming pattern called higher-order functions whereby you also pass functions to functions.

You can write custom HOCs or import them from libraries.

One example of an open source HOC is React Sortable HOC, whereby you pass a list component (based on `ul`) and receive an enhanced `ul` with sorting and drag and drop capabilities.

-  The key here would be to recall a time when you have used a HOC in your own project and to describe why it was the right pattern for the job
-  Learn more
  - This [open source repo](#) shows lots of different examples of what HOCs can look like

## What is portal?

React ordinarily has one mounting point - the HTML element you pass to `ReactDOM.render`. From here, React adds new elements to the page in a hierarchy.

Occasionally, you need to break out of this hierarchy.

Imagine a small About component with a button to open a modal. Because the modal "spills" out of the container, this not only feels unnatural, it can also be finicky to pull off because the About component might already have `overflow: hidden` set or a deliberate





## React interview questions

Portal and the `createPortal` function provide you with a way to render children in an additional mounting point (in addition to the main mounting point passed to `ReactDOM.render` ).

You're not too likely to read or write code using Portal in your own project.

Portal is mainly used when a parent component has an `overflow: hidden` or `z-index` , but you need the child to visually "break out" of its container.

Examples include modals, tooltips, and dialogs; however, we normally use third-party components for these things anyway, meaning we're unlikely to need to write Portal code ourselves.

-  Portals provide a first-class way to render children into a DOM node that exists outside the DOM hierarchy of the parent component
-  Learn more
  - [Portals](#)

## What are uncontrolled and controlled components?

A controlled component is an input component like an `input` , `textarea` or `select` whose value is controlled by React.

Conversely, an uncontrolled component manages its own state - the component is not controlled by React and is, therefore, "uncontrolled".

Imagine if you chuck a `textarea` on the page and start typing.

Anything you type will be stored in the `textarea` automatically and




## React interview questions


an example of an uncontrolled component.

To take control of this component in React, you would need to subscribe to the `textarea` `onChange` event and update a state variable (for example, one called `input` ) in response.

Now React is managing the `textarea` s value, you must also take responsibility for setting the `textarea` s value property also. This way, the content of the `textarea` can be updated by updating state. It's easy to imagine a function called `clearTextArea` that sets the input state variable to an empty string causing the `textarea` to clear.

-  The names "controlled component" and "uncontrolled component" are unnecessarily broad. More specific names would be "controlled *input* component" and "uncontrolled *input*

component" Narrowing your answer to focus on input components will ensure you answer the question eloquently..

-  Learn more
  - [React : Controlled vs. Uncontrolled components](#)

## Proven advice to secure your first developer job, with Austėja Kazlauskytė

Austėja is a new mom, law graduate, and “mom-and-pop” entrepreneur turned full-time frontend developer 🎉. This is the gutsy story about how Austėja found her passion for frontend development.



ALEX BOOKER

26 OCT 2020 • 12 MIN READ