# What is the difference between virtual and real DOM (React)?

**Grokking the Behavioral Interview**

Many candidates are rejected or down-leveled in technical interviews due to poor performance in behavioral or cultural fit interviews. Ace your interviews with this free course, where you will practice confidently tackling behavioral interview questions.

Get Free Course

**DOM** is the abstraction of the [HTML](https://www.educative.io/answers/catalog-html) of a web page. With the advent and popularity of dynamic web pages and SPAs, DOM has proven to be very inefficient and expensive when updating items.

Consider a list of numbers, as shown below:

              [1,2,3,4,5][1,2,3,4,5]

Suppose we wish to replace 33 with 66. For this, we'll create an entirely new list, as shown below:

              [1,2,6,4,5][1,2,6,4,5]

Another, more efficient way to do this is to go to the required index in the array and update it in place. This is a small example, but with thousands of nodes on a single page, the process of updating every node (re-rendering it) negatively affects the performance of the page.

### Virtual DOM Document Object Model (DOM)

The **virtual DOM** is the abstraction of the real DOM. In other words, it is the abstraction of an abstraction. A virtual DOM object is the same as a real DOM object, except that it is a lightweight copy. This means that it cannot manipulate on-screen elements. Moreover, upon any change of a property, it only updates the corresponding nodes and not the entire tree. That makes it a quick and efficient alternative.

### The process of updating in React

1. The [ReactDOM.render()](https://www.educative.io/answers/what-is-render-in-react-javascript" \t "_blank) renders the elements on the screen on the first load by creating the real and virtual DOM trees.
2. Any change to an element (such as a key press or button click) leads to a notification sent to the virtual nodes for a state change. If any property of the node is altered, it updates itself.
3. React compares the updated virtual DOM with the real DOM and updates the real DOM accordingly. This process is known asreconciliation. This is done using a heuristic algorithm known as theDiffing Algorithm.
4. The updated real DOM is rendered on the screen.

The illustration below outlines this process. The purple nodes represent the elements rendered on the first load, while the red node is the updated element due to user interaction. Initially, the real and virtual DOM trees are identical. Upon a change in an element, the virtual DOM tree is updated and compared to the real DOM, which is also updated.

### Difference between virtual and real DOM

The differences between virtual and real DOM are summarized in the table below:

|  |  |
| --- | --- |
| **Real DOM** | **Virtual DOM** |
| It is an abstraction of a page's HTML. | It is an abstraction of an HTML DOM. |
| It can manipulate on-screen elements. | It cannot manipulate on-screen elements. |
| Any change updates the entire DOM tree. | Any change only updates the relevant node in the tree. |
| Updating is slow and inefficient. | Updating is fast and efficient. |

# React Flux Concept

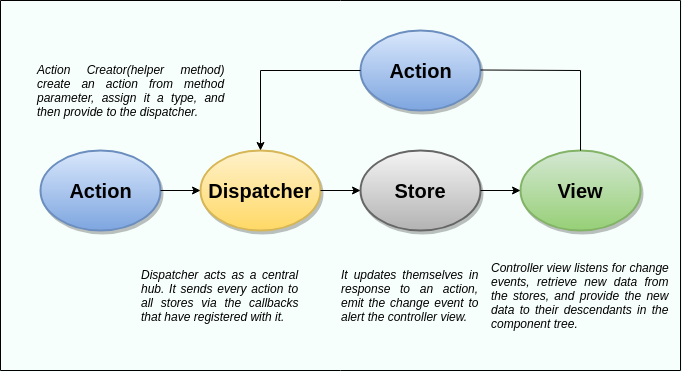
Flux is an application architecture that Facebook uses internally for building the client-side web application with React. It is not a library nor a framework. It is neither a library nor a framework. It is a kind of architecture that complements React as view and follows the concept of Unidirectional Data Flow model. It is useful when the project has dynamic data, and we need to keep the data updated in an effective manner. It reduces the runtime errors.

Flux applications have three major roles in dealing with data:

1. Dispatcher
2. Stores
3. Views (React components)

Here, you should not be confused with the Model-View-Controller (MVC) model. Although, Controllers exists in both, but Flux controller-views (views) found at the top of the hierarchy. It retrieves data from the stores and then passes this data down to their children. Additionally, action creators - dispatcher helper methods used to describe all changes that are possible in the application. It can be useful as a fourth part of the Flux update cycle.

## Structure and Data Flow



In Flux application, data flows in a single direction(unidirectional). This data flow is central to the flux pattern. The dispatcher, stores, and views are independent nodes with inputs and outputs. The actions are simple objects that contain new data and type property. Now, let us look at the various components of flux architecture one by one.

### Dispatcher

It is a central hub for the React Flux application and manages all data flow of your Flux application. It is a registry of callbacks into the stores. It has no real intelligence of its own, and simply acts as a mechanism for distributing the actions to the stores. All stores register itself and provide a callback. It is a place which handled all events that modify the store. When an action creator provides a new action to the dispatcher, all stores receive that action via the callbacks in the registry.

The dispatcher's API has five methods. These are:

|  |  |  |
| --- | --- | --- |
| **SN** | **Methods** | **Descriptions** |
| 1. | register() | It is used to register a store's action handler callback. |
| 2. | unregister() | It is used to unregisters a store's callback. |
| 3. | waitFor() | It is used to wait for the specified callback to run first. |
| 4. | dispatch() | It is used to dispatches an action. |
| 5. | isDispatching() | It is used to checks if the dispatcher is currently dispatching an action. |

### Stores

It primarily contains the application state and logic. It is similar to the model in a traditional MVC. It is used for maintaining a particular state within the application, updates themselves in response to an action, and emit the change event to alert the controller view.

### Views

It is also called as controller-views. It is located at the top of the chain to store the logic to generate actions and receive new data from the store. It is a React component listen to change events and receives the data from the stores and re-render the application.

### Actions

The dispatcher method allows us to trigger a dispatch to the store and include a payload of data, which we call an action. It is an action creator or helper methods that pass the data to the dispatcher.

### Advantage of Flux

* It is a unidirectional data flow model which is easy to understand.
* It is open source and more of a design pattern than a formal framework like MVC architecture.
* The flux application is easier to maintain.
* The flux application parts are decoupled.

**Introduction to Axios:**Axios, which is a popular library is mainly used to send asynchronous HTTP requests to REST endpoints. This library is very useful to perform CRUD operations.

1. This popular library is used to communicate with the backend. Axios supports the Promise API, native to JS ES6.
2. Using Axios we make API requests in our application. Once the request is made we get the data in Return, and then we use this data in our project.
3. This library is very popular among developers. You can check on GitHub and you will find 78k stars on it.

A polyfill allows you to use features that are not supported by a browser (or a specific browser version) by adding a fallback that mimics the desired behavior using supported APIs. You can use a polyfill when you want to use a JavaScript feature like String. padEnd , which isn't supported by older browsers.

Mounting

Mounting means putting elements into the DOM.

React has four built-in methods that gets called, in this order, when mounting a component:

constructor()

getDerivedStateFromProps()

render()

componentDidMount()

The render() method is required and will always be called, the others are optional and will be called if you define them.

constructor

The constructor() method is called before anything else, when the component is initiated, and it is the natural place to set up the initial state and other initial values.

The constructor() method is called with the props, as arguments, and you should always start by calling the super(props) before anything else, this will initiate the parent's constructor method and allows the component to inherit methods from its parent (React.Component).

Example:Get your own React.js Server

The constructor method is called, by React, every time you make a component:

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

render() {

return (

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

);

}

}

ReactDOM.render(<Header />, document.getElementById('root'));

### getDerivedStateFromProps

The getDerivedStateFromProps() method is called right before rendering the element(s) in the DOM.

This is the natural place to set the state object based on the initial props.

It takes state as an argument, and returns an object with changes to the state.

The example below starts with the favorite color being "red", but the getDerivedStateFromProps() method updates the favorite color based on the favcol attribute:

Mounting

Mounting means putting elements into the DOM.

React has four built-in methods that gets called, in this order, when mounting a component:

constructor()

getDerivedStateFromProps()

render()

componentDidMount()

The render() method is required and will always be called, the others are optional and will be called if you define them.

constructor

The constructor() method is called before anything else, when the component is initiated, and it is the natural place to set up the initial state and other initial values.

The constructor() method is called with the props, as arguments, and you should always start by calling the super(props) before anything else, this will initiate the parent's constructor method and allows the component to inherit methods from its parent (React.Component).

Example:Get your own React.js Server

The constructor method is called, by React, every time you make a component:

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

render() {

return (

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

);

}

}

ReactDOM.render(<Header />, document.getElementById('root'));

Lifecycle of Components

Each component in React has a lifecycle which you can monitor and manipulate during its three main phases.

The three phases are: Mounting, Updating, and Unmounting.

Mounting

Mounting means putting elements into the DOM.

React has four built-in methods that gets called, in this order, when mounting a component:

constructor()

getDerivedStateFromProps()

render()

componentDidMount()

The render() method is required and will always be called, the others are optional and will be called if you define them.

constructor

The constructor() method is called before anything else, when the component is initiated, and it is the natural place to set up the initial state and other initial values.

The constructor() method is called with the props, as arguments, and you should always start by calling the super(props) before anything else, this will initiate the parent's constructor method and allows the component to inherit methods from its parent (React.Component).

Example:Get your own React.js Server

The constructor method is called, by React, every time you make a component:

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

render() {

return (

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

);

}

}

ReactDOM.render(<Header />, document.getElementById('root'));

getDerivedStateFromProps

The getDerivedStateFromProps() method is called right before rendering the element(s) in the DOM.

This is the natural place to set the state object based on the initial props.

It takes state as an argument, and returns an object with changes to the state.

The example below starts with the favorite color being "red", but the getDerivedStateFromProps() method updates the favorite color based on the favcol attribute:

Example:

The getDerivedStateFromProps method is called right before the render method:

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

static getDerivedStateFromProps(props, state) {

return {favoritecolor: props.favcol };

}

render() {

return (

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

);

}

}

ReactDOM.render(<Header favcol="yellow"/>, document.getElementById('root'));

render

The render() method is required, and is the method that actually outputs the HTML to the DOM.

Example:

A simple component with a simple render() method:

class Header extends React.Component {

render() {

return (

<h1>This is the content of the Header component</h1>

);

}

}

ReactDOM.render(<Header />, document.getElementById('root'));

componentDidMount

The componentDidMount() method is called after the component is rendered.

This is where you run statements that requires that the component is already placed in the DOM.

Example:

At first my favorite color is red, but give me a second, and it is yellow instead:

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

componentDidMount() {

setTimeout(() => {

this.setState({favoritecolor: "yellow"})

}, 1000)

}

render() {

return (

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

);

}

}

ReactDOM.render(<Header />, document.getElementById('root'));

Updating

The next phase in the lifecycle is when a component is updated.

A component is updated whenever there is a change in the component's state or props.

React has five built-in methods that gets called, in this order, when a component is updated:

getDerivedStateFromProps()

shouldComponentUpdate()

render()

getSnapshotBeforeUpdate()

componentDidUpdate()

The render() method is required and will always be called, the others are optional and will be called if you define them.

getDerivedStateFromProps

Also at updates the getDerivedStateFromProps method is called. This is the first method that is called when a component gets updated.

This is still the natural place to set the state object based on the initial props.

The example below has a button that changes the favorite color to blue, but since the getDerivedStateFromProps() method is called, which updates the state with the color from the favcol attribute, the favorite color is still rendered as yellow:

Example:

If the component gets updated, the getDerivedStateFromProps() method is called:

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

static getDerivedStateFromProps(props, state) {

return {favoritecolor: props.favcol };

}

changeColor = () => {

this.setState({favoritecolor: "blue"});

}

render() {

return (

<div>

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

<button type="button" onClick={this.changeColor}>Change color</button>

</div>

);

}

}

ReactDOM.render(<Header favcol="yellow"/>, document.getElementById('root'));

shouldComponentUpdate

In the shouldComponentUpdate() method you can return a Boolean value that specifies whether React should continue with the rendering or not.

The default value is true.

The example below shows what happens when the shouldComponentUpdate() method returns false:

Example:

Stop the component from rendering at any update:

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

shouldComponentUpdate() {

return false;

}

changeColor = () => {

this.setState({favoritecolor: "blue"});

}

render() {

return (

<div>

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

<button type="button" onClick={this.changeColor}>Change color</button>

</div>

);

}

}

ReactDOM.render(<Header />, document.getElementById('root'));

Example:

Same example as above, but this time the shouldComponentUpdate() method returns true instead:

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

shouldComponentUpdate() {

return true;

}

changeColor = () => {

this.setState({favoritecolor: "blue"});

}

render() {

return (

<div>

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

<button type="button" onClick={this.changeColor}>Change color</button>

</div>

);

}

}

ReactDOM.render(<Header />, document.getElementById('root'));

render

The render() method is of course called when a component gets updated, it has to re-render the HTML to the DOM, with the new changes.

The example below has a button that changes the favorite color to blue:

Example:

Click the button to make a change in the component's state:

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

changeColor = () => {

this.setState({favoritecolor: "blue"});

}

render() {

return (

<div>

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

<button type="button" onClick={this.changeColor}>Change color</button>

</div>

);

}

}

ReactDOM.render(<Header />, document.getElementById('root'));

getSnapshotBeforeUpdate

In the getSnapshotBeforeUpdate() method you have access to the props and state before the update, meaning that even after the update, you can check what the values were before the update.

If the getSnapshotBeforeUpdate() method is present, you should also include the componentDidUpdate() method, otherwise you will get an error.

The example below might seem complicated, but all it does is this:

When the component is mounting it is rendered with the favorite color "red".

When the component has been mounted, a timer changes the state, and after one second, the favorite color becomes "yellow".

This action triggers the update phase, and since this component has a getSnapshotBeforeUpdate() method, this method is executed, and writes a message to the empty DIV1 element.

Then the componentDidUpdate() method is executed and writes a message in the empty DIV2 element:

Example:

Use the getSnapshotBeforeUpdate() method to find out what the state object looked like before the update:

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

componentDidMount() {

setTimeout(() => {

this.setState({favoritecolor: "yellow"})

}, 1000)

}

getSnapshotBeforeUpdate(prevProps, prevState) {

document.getElementById("div1").innerHTML =

"Before the update, the favorite was " + prevState.favoritecolor;

}

componentDidUpdate() {

document.getElementById("div2").innerHTML =

"The updated favorite is " + this.state.favoritecolor;

}

render() {

return (

<div>

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

<div id="div1"></div>

<div id="div2"></div>

</div>

);

}

}

ReactDOM.render(<Header />, document.getElementById('root'));

componentDidUpdate

The componentDidUpdate method is called after the component is updated in the DOM.

The example below might seem complicated, but all it does is this:

When the component is mounting it is rendered with the favorite color "red".

When the component has been mounted, a timer changes the state, and the color becomes "yellow".

This action triggers the update phase, and since this component has a componentDidUpdate method, this method is executed and writes a message in the empty DIV element:

Example:

The componentDidUpdate method is called after the update has been rendered in the DOM:

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

componentDidMount() {

setTimeout(() => {

this.setState({favoritecolor: "yellow"})

}, 1000)

}

componentDidUpdate() {

document.getElementById("mydiv").innerHTML =

"The updated favorite is " + this.state.favoritecolor;

}

render() {

return (

<div>

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

<div id="mydiv"></div>

</div>

);

}

}

ReactDOM.render(<Header />, document.getElementById('root'));

Unmounting

The next phase in the lifecycle is when a component is removed from the DOM, or unmounting as React likes to call it.

React has only one built-in method that gets called when a component is unmounted:

componentWillUnmount()

componentWillUnmount

The componentWillUnmount method is called when the component is about to be removed from the DOM.

Example:

Click the button to delete the header:

class Container extends React.Component {

constructor(props) {

super(props);

this.state = {show: true};

}

delHeader = () => {

this.setState({show: false});

}

render() {

let myheader;

if (this.state.show) {

myheader = <Child />;

};

return (

<div>

{myheader}

<button type="button" onClick={this.delHeader}>Delete Header</button>

</div>

);

}

}

class Child extends React.Component {

componentWillUnmount() {

alert("The component named Header is about to be unmounted.");

}

render() {

return (

<h1>Hello World!</h1>

);

}

}

ReactDOM.render(<Container />, document.getElementById('root'));

What are the advantages of ReactJS?

Entry

Below are the advantages of ReactJS:

Increases the application’s performance with Virtual DOM

JSX makes code easy to read and write

It renders both on the client and server-side

Easy to integrate with other frameworks (Angular, BackboneJS) since it is only a view library

Easy to write UI Test cases and integration with tools such as JEST.

How do you create refs in React?

How often does the React useState update? Why?

Name three advantages of using React Hooks.

Name two advantages of using React.js.

Outline the different stages of the React.js lifecycle.

Explain why class methods should be bound to class instances.

What happens if you attempt to update the state directly?

Explain whether Hooks replace higher-order components.

Name one advantage of using Flux.

5 crucial common advanced React.js interview questions and answers to look for

Here are five key common advanced React.js interview questions from the section above and the answers you should look for from your candidates.

1. How do you create refs in React?

Can your candidates explain that they must use React.createRef() to create refs in React.js? They should also understand that the ref is often assigned to an instance property when constructing a component.

2. Name two advantages of using React.js.

Candidates may mention several advantages of using React.js when responding to this question. For example, they may explain that the library helps them build high-quality user interfaces or that it permits them to write custom components.

3. Name one advantage of using Flux.

Your candidates may start by explaining that Flux is a JavaScript architecture that operates on a unidirectional data flow. But do they know Flux offers many advantages for projects with dynamic data? For example, can they explain that Flux helps ensure dynamic data is updated effectively?

4. Name three advantages of using React Hooks.

Three advantages of using React Hooks are that they enable developers to:

Enhance component tree readability

Share logic among different components

Effectively handle the setup of side effects

5. How often does the React useState update? Why?

Since developers use useState to enhance performance by creating queues, React doesn’t update changes immediately. Candidates should know that useState doesn’t implement changes to the state object directly; instead, the updates occur asynchronously.

8 advanced behavioral React.js interview questions

This section includes eight advanced behavioral React.js interview questions you can ask to find out which methods your candidates use when working on React.js projects.

image showing 8 advanced behavioral React.js interview questions

Which method would you use to handle events in React?

In which situation would you use refs in React?

Why would you use super constructors with props arguments?

How would you use validation on props?

Which method would you use to add attributes to components conditionally?

What methods would you use to check and improve slow app rendering in React?

In which situation would you use useMemo() in React?

How would you avoid binding in React?

5 crucial advanced behavioral React.js interview questions and answers to look for

Here are five of the most important advanced behavioral React.js interview questions from the section above and the answers you should look for from your candidates.

1. In which situation would you use refs in React?

Advanced candidates should understand that they can use React refs to access a DOM element. They may also explain that they would use refs to access an element they have created to change a child component’s value.

2. In which situation would you use useMemo() in React?

Do your candidates know that developers can use useMemo() to cache a variable’s value along with dependency lists? Can they explain that they would use useMemo() to help them avoid unnecessary re-renders? They may also say that useMemo() can be useful in situations where there are high processing amounts.

3. Why would you use super constructors with props arguments?

Candidates may explain that they pass props to super constructors to access and use this.props in the constructor. They may mention that when they implement a constructor() function within a React component, they use super() to call the parent constructor.

4. How would you avoid binding in React?

Candidates who have advanced React skills should be aware that they can use arrow functions in class properties to avoid binding in React. They may mention that class properties are a new feature and, to use them, a developer must enable transform-class-properties.

5. Which method would you use to handle events in React?

Can your candidates explain that to handle events in React, they would name them using camelCase (instead of lowercase)? They should be able to also explain that when they use JSX, they pass a function as the event handler (instead of a string).

28 advanced React.js interview questions about definitions and terms

Ask your candidates these advanced React.js interview questions about definitions and terms to discover if they have the knowledge to use the library efficiently.

image showing advanced React.js interview questions about definitions and terms

Explain what a higher-order component is.

Explain what a mounted component is.

Explain what useState is.

Explain what an event is in React.

What is a class component?

What is a component?

What is the difference between class and functional components?

What is a state object?

What is a props object?

How are state objects different from props objects?

Explain what MVC architecture is.

Name an architectural difference between React and Angular.

Explain what a controlled component is.

Explain what an uncontrolled component is.

How are controlled and uncontrolled components different?

Explain what React Hooks are.

Explain what three dots mean in React.

What are package managers in React.js?

Explain what prop drilling is.

Explain what StrictMode is.

Explain what the Shadow DOM is.

Explain what the virtual DOM is.

How is the Shadow DOM different from the virtual DOM?

Explain what the React.js lifecycle methods are.

Explain what a pure function is.

Explain what JSX is.

Explain what Flux architecture is.

What are bundlers in React.js?

5 crucial advanced React.js interview questions and answers related to definitions and terms

Here are five vital advanced React.js interview questions about definitions and terms, along with answers you should listen for from candidates.

1. Explain what React Hooks are.

Hooks are a feature that was introduced in React 16.8. They enable you to use state and other React features without having to write a class. They’re called Hooks because they allow you to “hook” into React features. One example is the useState Hook, which allows you to add React state to function components.

2. What is a state object?

A state object is a plain JavaScript object that developers use in React to show information on a component’s current properties. Developers can manage the state object in the component. Changing the state object causes the component to re-render.

3. What is a props object?

A props object takes the form of an ordinary object. Candidates should be able to explain that props objects abide by immutable properties, meaning that a component cannot change its own props object.

4. What is a class component?

A class component is a simple class that consists of several functions. It accepts props as arguments and returns React elements. Developers must create render functions to use class components and receive React elements.

5. What is a component?

A component is a reusable piece of code that developers can use to return React elements that will be rendered to a page. Components can return other components, as well as arrays, numbers, and strings.

8 tips for using advanced React.js interview questions

Before we conclude this article, we have listed eight tips in this section for using advanced React.js interview questions before, during, and after the interview.

image showing tips for using advanced React.js interview questions

1. Use your job description as a guide to help you build a list of advanced React.js interview questions

With a specific outline and description of the job you’re hiring for, you can easily build a set of interview questions. The job description will help you determine which criteria your candidates should meet.

It will also indicate the nice-to-have and must-have qualifications, so base your questions on the information in the job description.

2. Complete the skills-testing process before asking any advanced React.js interview questions

Skills testing can also help you build your list of advanced React.js interview questions. For example, once your candidates have completed the assessments, you may notice that they lack technical React.js skills or knowledge related to components. You can use this as an opportunity to include an interview question related to React.js components to learn more about your candidates’ skills.

3. Review the responsibilities of the role in more detail

Begin the interview by introducing yourself and explaining the role’s responsibilities in more detail. Reviewing the role will enable your candidates to match themselves to it and ask you questions about the open position.

For example, explain whether the position will be permanent or part-time, and describe the working conditions of the job.

4. Begin the interview with general or common interview questions

Learn about your candidates’ work-related experience and backgrounds by asking a few general or common interview questions. Find out how interested they are in the role, and give them another opportunity to evaluate how they would perform in the position if hired.

5. Be consistent when asking advanced React.js interview questions

Consistency is vital when asking candidates advanced React.js interview questions. Asking the same questions will help you compare candidates and keep the interview process fair.

You can ask follow-up questions, but your main list of questions should be identical for all candidates. This approach will prompt them to give more information when responding to your questions.

6. Always use open-ended advanced React.js interview questions

You’ll get more information from your candidates if you use open-ended advanced React.js interview questions, so avoid asking closed-ended questions to which your candidates can only reply “yes” or “no.”

If you do use closed-ended questions, ask a follow-up question afterward to get more information from your candidates.

7. Invert the interview process toward the end of the interview

Give your candidates the chance to ask their own questions at the end of the interview. Inverting the interview process will ensure that your candidates can determine whether they are fully suited to the open position.

When you invert the interview process, be prepared to answer their questions honestly, and ensure you have the information they need before the interview.

8. Let candidates know when they will hear back from you

Thank candidates for their time and let them know when you will contact them about the interview. This shows them your courtesy and enhances the candidate experience.

Start creating your list of advanced React.js interview questions to hire top talent

With this list of advanced React.js interview questions, hiring top talent will be a stress-free process. Use the questions you require and build your own list to evaluate your candidates.

Remember that skills testing is an essential part of the interview process that can make hiring easier.

Build and distribute skills assessments before the interview to hire the best talent for your organization. To accomplish this, you can use TestGorilla’s skills-testing platform, which has the most diverse range of skills tests.

Nothing’s stopping you now. Hire an exceptional developer with skills assessments and advanced React.js interview questions.

**What are the advantages of ReactJS?**

Entry

Below are the advantages of ReactJS:

1. Increases the application’s performance with Virtual DOM
2. JSX makes code easy to read and write
3. It renders both on the client and server-side
4. Easy to integrate with other frameworks (Angular, BackboneJS) since it is only a view library
5. Easy to write UI Test cases and integration with tools such as JEST.

**How does React work?**

Entry

React creates a virtual DOM. When state changes in a component it firstly runs a "diffing" algorithm, which identifies what has changed in the virtual DOM. The second step is reconciliation, where it updates the DOM with the results of the difference.

**What is the use of refs?**

Entry

**Refs** provide a way to access DOM nodes or React elements created in the render method. They should be avoided in most cases, however, they can be useful when we need direct access to the DOM element or an instance of a component.

There are a few good use cases for refs:

* Managing focus, text selection, or media playback.
* Triggering imperative animations.
* Integrating with third-party DOM libraries.

Refs are created using **React.createRef()** and attached to React elements via the **ref** attribute. Refs are commonly assigned to an instance property when a component is constructed so they can be referenced throughout the component.

class MyComponent extends React.Component {

constructor(props) {

super(props);

this.myRef = React.createRef(); }

render() {

return <div ref={this.myRef} />; }

}

**What are props in React?**

Entry

**Props** are inputs to a React component. They are single values or objects containing a set of values that are passed to React Components on creation using a naming convention similar to HTML-tag attributes. i.e, *They are data passed down from a parent component to a child component.*

The primary purpose of props in React is to provide the following component functionality:

1. Pass custom data to your React component.
2. Trigger **state** changes.
3. Use via **this.props.reactProp** inside component's **render()** method.

For example, let us create an element with **reactProp** property,

<Element reactProp = "1" />

This **reactProp** (or whatever you came up with) the name then becomes a property attached to React's native props object which originally already exists on all components created using React library.

props.reactProp;

**What is Context API in ReactJS?**

Entry

Context provides a way to pass data through the component tree without having to pass props down manually at every level.

Context is designed to share data that can be considered “gl=obal” for a tree of React components, such as the current authenticated user, theme, or preferred language. Using context, we can avoid passing props through intermediate elements.

// Context lets us pass a value deep into the component tree

// without explicitly threading it through every component.

// Create a context for the current theme (with "light" as the default).

const ThemeContext = React.createContext('light');

class App extends React.Component {

render() {

// Use a Provider to pass the current theme to the tree below.

// Any component can read it, no matter how deep it is.

// In this example, we're passing "dark" as the current value.

return (

<ThemeContext.Provider value="dark">

<Toolbar />

</ThemeContext.Provider>

);

}

}

// A component in the middle doesn't have to

// pass the theme down explicitly anymore.

function Toolbar() {

return (

<div>

<ThemedButton />

</div>

);

}

class ThemedButton extends React.Component {

// Assign a contextType to read the current theme context.

// React will find the closest theme Provider above and use its value.

// In this example, the current theme is "dark".

static contextType = ThemeContext;

render() {

return <Button theme={this.context} />;

}

}

**What are React Hooks?**

Entry

**Hooks** are a new addition to React 16.8. They let you use state and other React features without writing a class.

With Hooks, you can extract stateful logic from a component so it can be tested independently and reused. Hooks allow you to reuse stateful logic without changing your component hierarchy. This makes it easy to share Hooks among many components or with the community.

**What are the major features of ReactJS?**

Entry

The major features of ReactJS are as follows,

* It uses **VirtualDOM** instead RealDOM considering that RealDOM manipulations are expensive.
* Supports **server-side rendering**
* Follows **Unidirectional** data flow or data binding
* Uses **reusable/composable** UI components to develop the view

**How would you write an inline style in React?**

Entry

For example:

<div style={{ height: 10 }}>

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

Entry

* The **state** is a data structure that starts with a default value when a Component mounts. It may be mutated across time, mostly as a result of user events.
* **Props** (short for properties) are a Component's configuration. They are received from above and immutable as far as the Component receiving them is concerned. A Component cannot change its props, but it is responsible for putting together the props of its child Components. Props do not have to just be data - callback functions may be passed in as props.

**What is JSX?**

Entry

JSX is a syntax notation for **JavaScript XML** (XML-like syntax extension to ECMAScript). It stands for JavaScript XML. It provides the expressiveness of JavaScript along with HTML-like template syntax.

For example, the below text inside the **h1** tag return as a javascript function to the render function,

render(){

return(

<div>

<h1> Welcome to React world!!</h1>

</div>

);

}

**What are the differences between a Class component and a Functional component?**

Junior

**Class Components**

* Class-based Components use ES6 class syntax. It can make use of the lifecycle methods.
* Class components extend from**React.Component**.
* Here you have to use this keyword to access the props and functions that you declare inside the class components.

**Functional Components**

* Functional Components are simpler compared to class-based functions.
* Functional Components mainly focus on the UI of the application, not on the behavior.
* To be more precise these are basically rendered functions in the class component.
* Functional Components can have state and mimic lifecycle events using Reach Hooks

**What is the difference between a Presentational component and a Container component?**

Junior

* **Presentational components** are concerned with *how things look*. They generally receive data and callbacks exclusively via props. These components rarely have their own state, but when they do it generally concerns the UI state, as opposed to the data state.
* **Container components** are more concerned with *how things work*. These components provide the data and behavior to presentational or other container components. They call Flux actions and provide these as callbacks to the presentational components. They are also often stateful as they serve as data sources.

**What's the difference between a Controlled component and an Uncontrolled one in React?**

Junior

This relates to stateful DOM components (form elements) and the React docs explain the difference:

* A [Controlled Component](https://facebook.github.io/react/docs/forms.html#controlled-components) is one that takes its current value through **props** and notifies changes through callbacks like **onChange**. A parent component "controls" it by handling the callback and managing its own state and passing the new values as props to the controlled component. You could also call this a "dumb component".
* An [Uncontrolled Component](https://facebook.github.io/react/docs/uncontrolled-components.html) is one that stores its own state internally, and you query the DOM using a **ref** to find its current value when you need it. This is a bit more like traditional HTML.

Most native React form components support both controlled and uncontrolled usage:

// Controlled:

<input type="text" value={value} onChange={handleChange} />

// Uncontrolled:

<input type="text" defaultValue="foo" ref={inputRef} />

// Use `inputRef.current.value` to read the current value of <input>

**What does it mean for a component to be mounted in React?**

Junior

It has a corresponding element created in the DOM and is connected to that.

Mention some limitations of React?

Junior

React is just a view library, not a full-blown framework

There is a learning curve for beginners who are new to web development.

Integrating React.js into a traditional MVC framework requires some additional configuration

The code complexity increases with inline templating and JSX.

Too many smaller components lead to over-engineering or boilerplate

What are Stateless components in React?

Junior

If the behavior is independent of its state then it can be a stateless component. You can use either a function or a class for creating stateless components. But unless you need to use a lifecycle hook in your components, you should go for stateless functional components.

Stateful/Container/Smart component:

class Main extends Component {

constructor() {

super()

this.state = {

books: []

}

}

render() {

<BooksList books={this.state.books} />

}

}

Stateless/Presentational/Dumb component:

const BooksList = ({books}) => {

return (

<ul>

{books.map(book => {

return <li>book</li>

})}

</ul>

)

}

There are a lot of benefits if you decide to use stateless functional components here; they are:

easy to write, understand, test, and

you can avoid this keyword altogether.

What is the purpose of the callback function as an argument of setState?

Junior

The callback function is invoked when setState finished and the component gets rendered. Since setState is asynchronous the callback function is used for any post-action.

Note: It is recommended to use the lifecycle method rather than this callback function.

setState({name: 'protechstack'}, () => console.log('The name has updated and component re-rendered'));

What are the advantages of using React Hooks?

Junior

Primarily, hooks in general enable the extraction and reuse of stateful logic that is common across multiple components without the burden of higher order components or render props. Hooks allow us to easily manipulate the state of our functional components without needing to convert them into class components.

Hooks don’t work inside classes (because they let you use React without classes). By using them, we can totally avoid using lifecycle methods, such as componentDidMount, componentDidUpdate, componentWillUnmount. Instead, we will use built-in hooks like useEffect .

What are portals in React and when do we need them?

Junior

Portals provide a first-class way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.

Sometimes it’s useful to insert a child into a different location in the DOM:

render() {

// React does \*not\* create a new div. It renders the children into `domNode`.

// `domNode` is any valid DOM node, regardless of its location in the DOM.

return ReactDOM.createPortal(

this.props.children,

domNode );

}

A typical use case for portals is when a parent component has an overflow: hidden or z-index style, but you need the child to visually “break out” of its container.

What happens during the lifecycle of a React component?

Junior

At the highest level, React components have lifecycle events that fall into three general categories:

Initialization

State/Property Updates

Destruction

What are inline conditional expressions in ReactJS?

Junior

You can use either if statements or ternary expressions which are available from JS to conditionally render expressions.

Apart from these approaches, you can also embed any expressions in JSX by wrapping them in curly braces and then followed by JS logical operator(&&).

if(this.state.mode === 'view') {

return (

<button onClick={this.handleEdit}>

Edit

</button>

);

} else {

return (

<button onClick={this.handleSave}>

Save

</button>

);

}

// or

{

view

? null

: (

<p>

<input

onChange={this.handleChange}

value={this.state.inputText} />

</p>

)

}

### [Gatsby: The Best React-Based Framework](https://www.gatsbyjs.com/)

[Gatsby](https://www.gatsbyjs.com/)

[https://www.gatsbyjs.com](https://www.gatsbyjs.com/)

**Gatsby** is a React-based open source framework with performance, scalability and security built-in. Collaborate, build and deploy 1000x faster on Netlify.