likezninjaz / **react-interview-questions**

Looking for React.js jobs? Here are the most popular React Interview Questions and Answers which are most likely to be asked by the interviewer.

6 stars 1 fork

Star Watch

Code Issues Pull requests Actions Projects Wiki Security

master

likezninjaz Update README.md ... ✓ on Jul 18, 2020 16

View code

React Interview Questions

Here are the most popular questions asked at interviews of front-end developers on React.js. The topics include the basics of JavaScript and web technologies and a deep

README.md

JavaScript:

▼ What types of data exist in JavaScript?

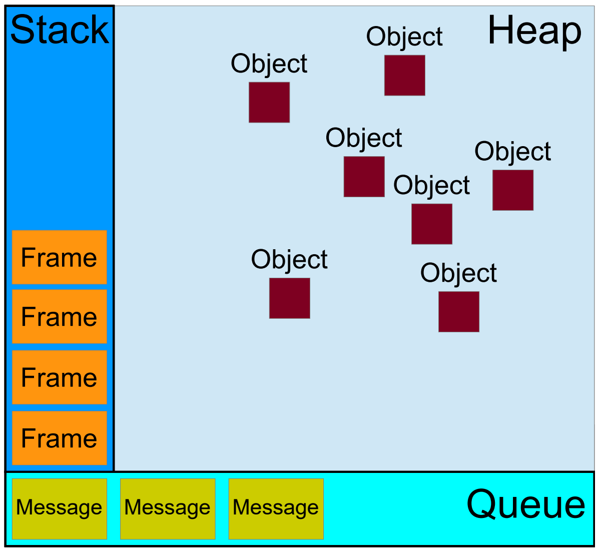
- «**number**» - The number type represents both integer and floating point numbers.
- «**BigInt**» - BigInt type was recently added to the language to represent integers of arbitrary length.
- «**string**» - A string in JavaScript must be surrounded by quotes.
- «**boolean**» - The boolean type has only two values: true and false.
- «**null**» - The special null value does not belong to any of the types described above. It forms a separate type of its own which contains only the null value.
- «**undefined**» - The special value undefined also stands apart. It makes a type of its own, just like null. The meaning of undefined is “value is not assigned”.

- **«object»** - The object type is special. All other types are called "primitive" because their values can contain only a single thing (be it a string or a number or whatever). In contrast, objects are used to store collections of data and more complex entities.
- **symbol** - The symbol type is used to create unique identifiers for objects. We have to mention it here for the sake of completeness, but also postpone the details till we know objects.

Source: javascript.info

▼ Whats is event loop in JavaScript and how it works?

JavaScript has a concurrency model based on an event loop, which is responsible for executing the code, collecting and processing events, and executing queued sub-tasks. This model is quite different from models in other languages like C and Java.



Stack

Function calls form a stack of frames.

```
function foo(b) {
  let a = 10
  return a + b + 11
}

function bar(x) {
  let y = 3
  return foo(x * y)
}

console.log(bar(7)) //returns 42
```

When calling bar, a first frame is created containing bar's arguments and local variables. When bar calls foo, a second frame is created and pushed on top of the first one containing foo's arguments and local variables. When foo returns, the top frame element is popped out of the stack (leaving only bar's call frame). When bar returns, the stack is empty.

Heap

Objects are allocated in a heap which is just a name to denote a large (mostly unstructured) region of memory.

Queue

A JavaScript runtime uses a message queue, which is a list of messages to be processed. Each message has an associated function which gets called in order to handle the message. The processing of functions continues until the stack is once again empty. Then, the event loop will process the next message in the queue (if there is one).

Event loop

The event loop got its name because of how it's usually implemented, which usually resembles:

```
while (queue.waitForMessage()) {
  queue.processNextMessage()
}
```

queue.waitForMessage() waits synchronously for a message to arrive (if one is not already available and waiting to be handled).

Source: MDN web docs

▼ What is closure in JavaScript?

A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, a closure gives you access to an outer function’s scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

Every closure has three scopes:

- Local Scope (Own scope)
- Outer Functions Scope

- Global Scope

A common mistake is not realizing that, in the case where the outer function is itself a nested function, access to the outer function's scope includes the enclosing scope of the outer function—effectively creating a chain of function scopes. To demonstrate, consider the following example code.

```
// global scope
var e = 10;
function sum(a){
  return function(b){
    return function(c){
      // outer functions scope
      return function(d){
        // local scope
        return a + b + c + d + e;
      }
    }
  }
}

console.log(sum(1)(2)(3)(4)); // log 20

// You can also write without anonymous functions:

// global scope
var e = 10;
function sum(a){
  return function sum2(b){
    return function sum3(c){
      // outer functions scope
      return function sum4(d){
        // local scope
        return a + b + c + d + e;
      }
    }
  }
}

var s = sum(1);
var s1 = s(2);
var s2 = s1(3);
var s3 = s2(4);
console.log(s3) //log 20
```

In the example above, there's a series of nested functions, all of which have access to the outer functions' scope. In this context, we can say that closures have access to all outer function scopes.

Source: MDN web docs

▼ How does the “this” keyword work?

A function's this keyword behaves a little differently in JavaScript compared to other languages. It also has some differences between strict mode and non-strict mode.

- **Global context:** In the global execution context (outside of any function), this refers to the global object whether in strict mode or not.
- **Function context:** Inside a function, the value of this depends on how the function is called.

  Since the following code is not in strict mode, and because the value of this is not set by the call, this will default to the global object, which is window in a browser.

  ```
  function f1() {
    return this;
  }

  // In a browser:
  f1() === window; // true

  // In Node:
  f1() === globalThis; // true
  ```

  In strict mode, however, if the value of this is not set when entering an execution context, it remains as undefined.

  ```
  function f2() {
    'use strict'; // see strict mode
    return this;
  }

  f2() === undefined; // true
  ```

- **Class context:** The behavior of this in classes and functions is similar, since classes are functions under the hood. But there are some differences and caveats. Within a class constructor, this is a regular object. All non-static methods within the class are added to the prototype of this.
- **Derived classes:** Unlike base class constructors, derived constructors have no initial this binding. Calling super() creates a this binding within the constructor and essentially has the effect of evaluating the following line of code, where Base is the inherited class. Derived classes must not return before calling super(), unless they

return an Object or have no constructor at all. Referring to this before calling super() will throw an error.

Source: MDN web docs

▼ What are promises in JavaScript?

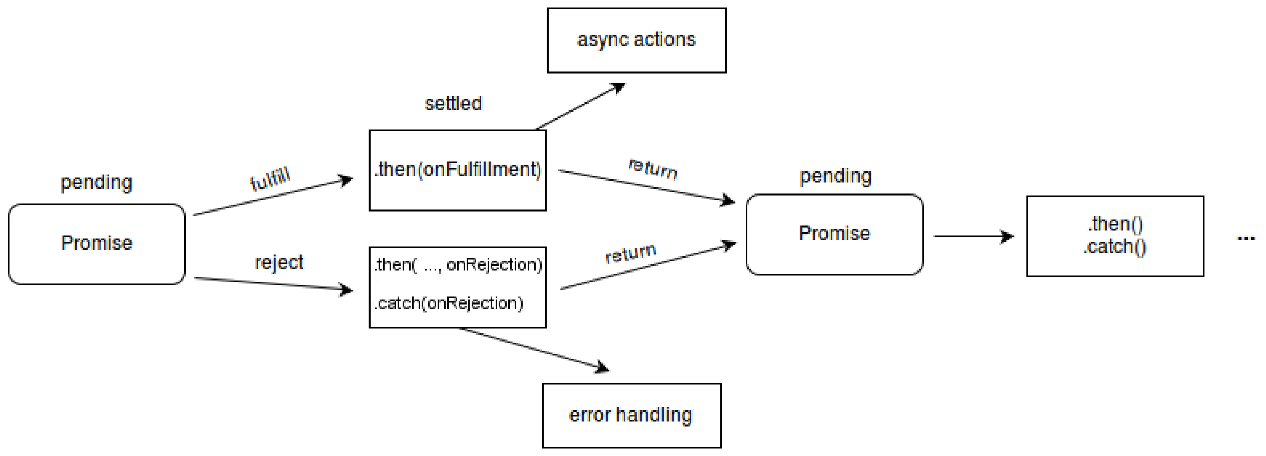
The Promise object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.

A Promise is in one of these states:

- *pending:* initial state, neither fulfilled nor rejected.
- *fulfilled:* meaning that the operation completed successfully.
- *rejected:* meaning that the operation failed.

A pending promise can either be fulfilled with a value, or rejected with a reason (error). When either of these options happens, the associated handlers queued up by a promise's then method are called.

As the Promise.prototype.then() and Promise.prototype.catch() methods return promises, they can be chained.



The methods promise.then(), promise.catch(), and promise.finally() are used to associate further action with a promise that becomes settled. These methods also return a newly generated promise object, which can optionally be used for chaining; for example, like this:

```
const myPromise =
  (new Promise(myExecutorFunc))
  .then(handleFulfilledA,handleRejectedA)
  .then(handleFulfilledB,handleRejectedB)
  .then(handleFulfilledC,handleRejectedC);
```

```
// or, perhaps better ...

const myPromise =
  (new Promise(myExecutorFunc))
  .then(handleFulfilledA)
  .then(handleFulfilledB)
  .then(handleFulfilledC)
  .catch(handleRejectedAny);
```

Source: MDN web docs

React:

▼ What component life cycle methods exist in React?

- **render()** — The render() method is the only required method in a class component. When called, it should examine this.props and this.state and return one of the following types: React elements, Arrays and fragments, Portals, String and numbers, Booleans or null.
  The render() function should be pure, meaning that it does not modify component state, it returns the same result each time it's invoked, and it does not directly interact with the browser.

- **constructor()** - The constructor for a React component is called before it is mounted. When implementing the constructor for a React.Component subclass, you should call super(props) before any other statement. Otherwise, this.props will be undefined in the constructor, which can lead to bugs.
  Typically, in React constructors are only used for two purposes: Initializing local state by assigning an object to this.state. Binding event handler methods to an instance. Constructor is the only place where you should assign this.state directly. In all other methods, you need to use this.setState() instead.

- **componentDidMount()** - is invoked immediately after a component is mounted (inserted into the tree). Initialization that requires DOM nodes should go here. If you need to load data from a remote endpoint, this is a good place to instantiate the network request.
  This method is a good place to set up any subscriptions. If you do that, don't forget to unsubscribe in componentWillUnmount().

- **componentDidUpdate(prevProps, prevState, snapshot)** - is invoked immediately after updating occurs. This method is not called for the initial render.

Use this as an opportunity to operate on the DOM when the component has been updated. This is also a good place to do network requests as long as you compare the current props to previous props (e.g. a network request may not be necessary if the props have not changed).

- **componentWillUnmount()** - is invoked immediately before a component is unmounted and destroyed. Perform any necessary cleanup in this method, such as invalidating timers, canceling network requests, or cleaning up any subscriptions that were created in componentDidMount().

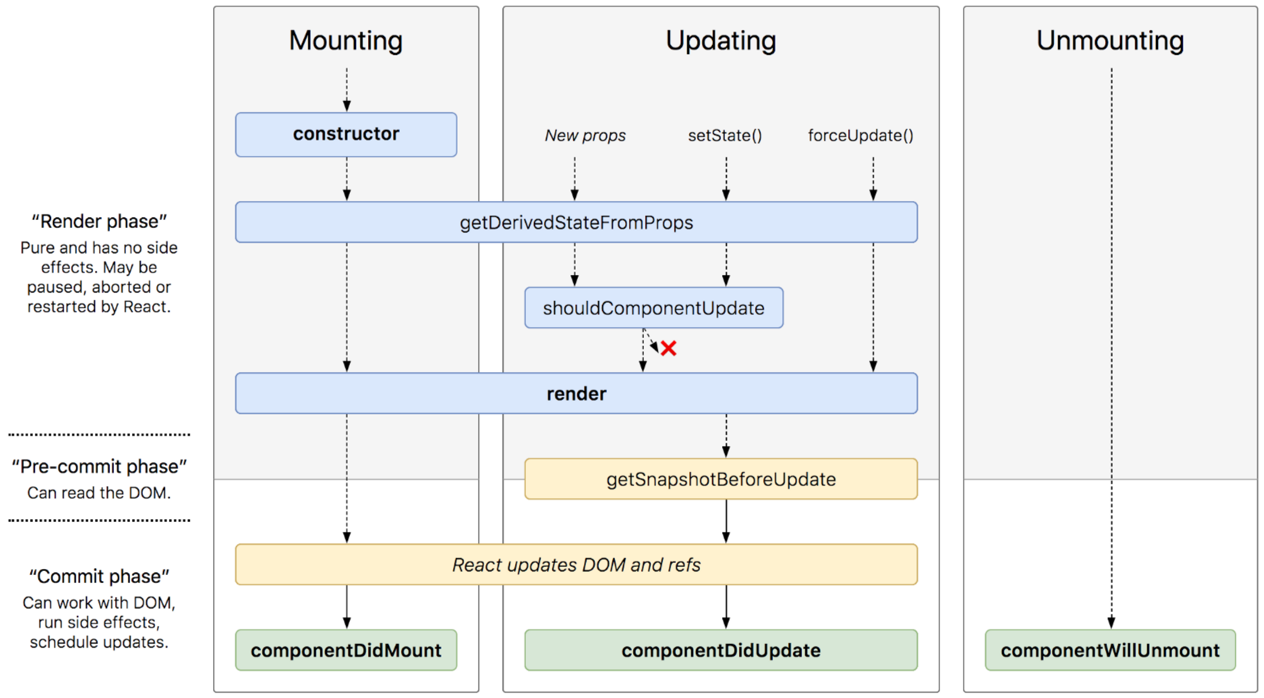
- **shouldComponentUpdate(nextProps, nextState)** - is invoked before rendering when new props or state are being received. Defaults to true. This method is not called for the initial render or when forceUpdate() is used. This method only exists as a performance optimization.

- **static getDerivedStateFromProps(props, state)** - is invoked right before calling the render method, both on the initial mount and on subsequent updates. It should return an object to update the state, or null to update nothing.
  This method exists for rare use cases where the state depends on changes in props over time.

- **getSnapshotBeforeUpdate(prevProps, prevState)** - is invoked right before the most recently rendered output is committed to e.g. the DOM. It enables your component to capture some information from the DOM (e.g. scroll position) before it is potentially changed. Any value returned by this lifecycle will be passed as a parameter to componentDidUpdate().

- **static getDerivedStateFromError(error)** - This lifecycle is invoked after an error has been thrown by a descendant component. It receives the error that was thrown as a parameter and should return a value to update state. getDerivedStateFromError() is called during the “render” phase, so side-effects are not permitted. For those use cases, use componentDidCatch() instead.

- **componentDidCatch(error, info)** - This lifecycle is invoked after an error has been thrown by a descendant component. It receives two parameters: error - The error that was thrown, info - An object with a componentStack key containing information about which component threw the error. It should be used for things like logging errors.



Source: reactjs.org

▼ What is Context in React?

Context is designed to share data that can be considered “global” 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} />;
  }
}
```

Context is primarily used when some data needs to be accessible by many components at different nesting levels. Apply it sparingly because it makes component reuse more difficult.

API:

- **React.createContext** - creates a Context object. When React renders a component that subscribes to this Context object it will read the current context value from the closest matching Provider above it in the tree.
- **Context.Provider** - every Context object comes with a Provider React component that allows consuming components to subscribe to context changes.
- **Class.contextType** - the contextType property on a class can be assigned a Context object created by React.createContext(). This lets you consume the nearest current value of that Context type using this.context. You can reference this in any of the lifecycle methods including the render function.
- **Context.Consumer** - a React component that subscribes to context changes. This lets you subscribe to a context within a function component. Requires a function as a child. The function receives the current context value and returns a React node. The value argument passed to the function will be equal to the value prop of the closest Provider for this context above in the tree. If there is no Provider for this context above, the value argument will be equal to the defaultValue that was passed to createContext().
- **Context.displayName** - Context object accepts a displayName string property. React DevTools uses this string to determine what to display for the context.

Source: reactjs.org

▼ What is the Virtual DOM?

The virtual DOM (VDOM) is a programming concept where an ideal, or “virtual”, representation of a UI is kept in memory and synced with the “real” DOM by a library such as ReactDOM. This process is called reconciliation.

In React world, the term “virtual DOM” is usually associated with React elements since they are the objects representing the user interface. React, however, also uses internal objects called “fibers” to hold additional information about the component tree. They may also be considered a part of “virtual DOM” implementation in React.

Source: reactjs.org

▼ Why do you need the key attribute when rendering lists?

Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside the array to give the elements a stable identity

The best way to pick a key is to use a string that uniquely identifies a list item among its siblings. Most often you would use IDs from your data as keys

When you don’t have stable IDs for rendered items, you may use the item index as a key as a last resort. We don’t recommend using indexes for keys if the order of items may change. This can negatively impact performance and may cause issues with component state.

Source: reactjs.org

▼ How does "children" prop work?

Some components don’t know their children ahead of time. This is especially common for components like Sidebar or Dialog that represent generic “boxes”. We recommend that such components use the special children prop to pass children elements directly into their output:

```
function FancyBorder(props) {
  return (
    <div className={'FancyBorder FancyBorder-' + props.color}>
      {props.children}
    </div>
  );
}
```

This lets other components pass arbitrary children to them by nesting the JSX:

```
function WelcomeDialog() {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        Welcome
      </h1>
      <p className="Dialog-message">
        Thank you for visiting our spacecraft!
      </p>
    </FancyBorder>
  );
}
```

Anything inside the JSX tag gets passed into the FancyBorder component as a children prop. Since FancyBorder renders {props.children} inside a div, the passed elements appear in the final output.

Source: reactjs.org

▼ What is the difference between controlled and uncontrolled components?

In HTML, form elements such as input, textarea, and select typically maintain their own state and update it based on user input. In React, mutable state is typically kept in the state property of components, and only updated with setState().

We can combine the two by making the React state be the “single source of truth”. Then the React component that renders a form also controls what happens in that form on subsequent user input. An input form element whose value is controlled by React in this way is called a **“controlled component”**.

To write an **uncontrolled component**, instead of writing an event handler for every state update, you can use a ref to get form values from the DOM.

Since an uncontrolled component keeps the source of truth in the DOM, it is sometimes easier to integrate React and non-React code when using uncontrolled components. It can also be slightly less code if you want to be quick and dirty. Otherwise, you should usually use controlled components.

Source: reactjs.org, reactjs.org

▼ What is PureComponent?

React.PureComponent is similar to React.Component. The difference between them is that React.Component doesn't implement shouldComponentUpdate(), but React.PureComponent implements it with a shallow prop and state comparison.

If your React component's render() function renders the same result given the same props and state, you can use React.PureComponent for a performance boost in some cases.

React.PureComponent's shouldComponentUpdate() only shallowly compares the objects. If these contain complex data structures, it may produce false-negatives for deeper differences. Only extend PureComponent when you expect to have simple props and state, or use forceUpdate() when you know deep data structures have changed. Or, consider using immutable objects to facilitate fast comparisons of nested data. Furthermore, React.PureComponent's shouldComponentUpdate() skips prop updates for the whole component subtree. Make sure all the children components are also "pure".

Source: reactjs.org

▼ What are the Higher-Order Components?

A higher-order component (HOC) is an advanced technique in React for reusing component logic. HOCs are not part of the React API, per se. They are a pattern that emerges from React's compositional nature.

Concretely, a higher-order component is a function that takes a component and returns a new component.

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

HOCs are common in third-party React libraries, such as Redux's connect and Relay's createFragmentContainer.

Source: reactjs.org

▼ What are basic React Hooks?

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class. Hooks don't work inside classes. But you can use them instead of writing classes.

A Hook is a special function that lets you "hook into" React features. For example, **useState** is a Hook that lets you add React state to function components:

```
import React, { useState } from 'react';

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

It declares a “state variable”. Our variable is called count but we could call it anything else, like banana. This is a way to “preserve” some values between the function calls — useState is a new way to use the exact same capabilities that this.state provides in a class. Normally, variables “disappear” when the function exits but state variables are preserved by React. The only argument to the useState() Hook is the initial state. Unlike with classes, the state doesn’t have to be an object. We can keep a number or a string if that’s all we need. In our example, we just want a number for how many times the user clicked, so pass 0 as initial state for our variable. (If we wanted to store two different values in state, we would call useState() twice.). It returns a pair of values: the current state and a function that updates it. This is why we write const [count, setCount] = useState().

The **Effect Hook** lets you perform side effects in function components:

```
 import React, { useState, useEffect } from 'react';

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

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
      </div>
    );
  }
```

By using this Hook, you tell React that your component needs to do something after render. React will remember the function you passed (we'll refer to it as our "effect"), and call it later after performing the DOM updates. In this effect, we set the document title, but we could also perform data fetching or call some other imperative API. Placing useEffect inside the component lets us access the count state variable (or any props) right from the effect. We don't need a special API to read it — it's already in the function scope. Hooks embrace JavaScript closures and avoid introducing React-specific APIs where JavaScript already provides a solution. By default, it runs both after the first render and after every update. Instead of thinking in terms of "mounting" and "updating", you might find it easier to think that effects happen "after render". React guarantees the DOM has been updated by the time it runs the effects.

useContext

```
const value = useContext(MyContext);
```

Accepts a context object (the value returned from React.createContext) and returns the current context value for that context. The current context value is determined by the value prop of the nearest above the calling component in the tree. When the nearest above the component updates, this Hook will trigger a rerender with the latest context value passed to that MyContext provider. Even if an ancestor uses React.memo or shouldComponentUpdate, a rerender will still happen starting at the component itself using useContext.

Source: reactjs.org

Web technologies:

▶ What is HTTP?

Releases

No releases published

Packages

No packages published