Frequently asked: React JS Interview Questions and Answers

Vigo Webs Follow
Sep 23, 2018 · 12 min read

Q1. How React works? How Virtual-DOM works in React?

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 diff.

The HTML DOM is always tree-structured—which is allowed by the structure of HTML document. The DOM trees are huge nowadays because of large apps. Since we are more and more pushed towards dynamic web apps (Single Page Applications—SPAs), we need to modify the DOM tree incessantly and a lot. And this is a real performance and development pain.

The Virtual DOM is an abstraction of the HTML DOM. It is lightweight and detached from the browser-specific implementation details. It is not invented by React but it uses it and provides it for free. `ReactElements` lives in the virtual DOM. They make the basic nodes here. Once we defined the elements, `ReactElements` can be render into the "real" DOM.

Whenever a `ReactComponent` is changing the state, diff algorithm in React runs and identifies what has changed. And then it updates the DOM with the results of diff. The point is - it's done faster than it would be in the regular DOM.

Q2. What is JSX?

JSX is a syntax extension to JavaScript and comes with the full power of JavaScript. JSX produces React "elements". You can embed any JavaScript expression in JSX by wrapping it in curly braces. After compilation, JSX expressions become regular JavaScript objects. This means that you can use JSX inside of if statements and for loops, assign it to variables, accept it as arguments, and return it from functions. Eventhough React does not require JSX, it is the recommended way of describing our UI in React app.

For example, below is the syntax for a basic element in React with JSX and its equivalent without it.

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

Equivalent of the above using `React.createElement`

```
const element = React.createElement(
  'h1',
  {"className": "greeting"},
  'Hello, world!'
);
```

Q3. What is `React.createClass` ?

`React.createClass` allows us to generate component "classes." But with ES6, React allows us to implement component classes that use ES6 JavaScript classes. The end result is the same -- we have a component class. But the style is different. And one is using a "custom" JavaScript

class system (createClass) while the other is using a "native" JavaScript class system.

When using React's `createClass()` method, we pass in an object as an argument. So we can write a component using `createClass` that looks like this:

```
import React from 'react';

const Contacts = React.createClass({
  render() {
    return (
      <div></div>
    );
  }
});

export default Contacts;
```

Using an ES6 class to write the same component is a little different. Instead of using a method from the `react` library, we extend an ES6 class that the library defines, `Component`.

```
import React from 'react';

class Contacts extends React.Component({
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div></div>
    );
  }
}

export default Contacts;
```

`constructor()` is a special function in a JavaScript class. JavaScript invokes `constructor()` whenever an object is created via a class.

Q4. What is `ReactDOM` and what is the difference between `ReactDOM` and `React`?

Prior to v0.14, all `ReactDOM` functionality was part of `React`. But later, `React` and `ReactDOM` were split into two different libraries.

As the name implies, `ReactDOM` is the glue between React and the DOM. Often, we will only use it for one single thing: mounting with `ReactDOM`. Another useful feature of `ReactDOM` is `ReactDOM.findDOMNode()` which we can use to gain direct access to a DOM element.

For everything else, there's `React`. We use React to define and create our elements, for lifecycle hooks, etc. i.e. the guts of a React application.

Q5. What are the differences between a class component and functional component?

Class components allows us to use additional features such as local state and lifecycle hooks. Also, to enable our component to have direct access to our store and thus holds state.

When our component just receives props and renders them to the page, this is a 'stateless component', for which a pure function can be used. These are also called dumb components or presentational components.

From the previous question, we can say that our `Booklist` component is functional components and are stateless.

```
// BookList.js

import React from "react";

const Booklist = books => (
  <ul>
    {books.map(({ title, author }) =>

•  {title}-{author}

    )}
  </ul>
)
```

On the other hand, the `BookListContainer` component is a class component.

Q6. What is the difference between state and props?

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. Props are how components talk to each other. They are received from above component 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.

There is also the case that we can have default props so that props are set even if a parent component doesn't pass props down.

```
Class SearchBar extends Component {
  constructor(props) {
    super(props);
    this.state = { term: '' };
  }
  render() {
    return (
      <div className="search-bar">
      <input
        value={this.state.term}
        onChange={event => this.onInputChange(event.target.value)} />
      </div>
    );
  }
  onInputChange(term) {
    this.setState({term});
    this.props.onSearchTermChange(term);
  }
}
```

Props and State do similar things but are used in different ways. The majority of our components will probably be stateless. Props are used to pass data from parent to child or by the component itself. They are immutable and thus will not be changed. State is used for mutable data, or data that will change. This is particularly useful for user input.

Q7. What are controlled components?

In HTML, form elements such as `<input>`, `<textarea>`, and `<select>` typically maintain their own state and update it based on user input. When a user submits a form the values from the aforementioned elements are sent with the form. With React it works differently. The component containing the form will keep track of the

value of the input in it's state and will re-render the component each time the callback function e.g. `onChange` is fired as the state will be updated. A form element whose value is controlled by React in this way is called a "controlled component".

With a controlled component, every state mutation will have an associated handler function. This makes it straightforward to modify or validate user input.

Q8. What is a higher order component?

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

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

HOC's allow you to reuse code, logic and bootstrap abstraction. HOCs are common in third-party React libraries. The most common is probably Redux's connect function. Beyond simply sharing utility libraries and simple composition, HOCs are the best way to share behavior between React Components. If you find yourself writing a lot of code in different places that does the same thing, you may be able to refactor that code into a reusable HOC.

Q9. What is `create-react-app` ?

`create-react-app` is the official CLI (Command Line Interface) for React to create React apps with no build configuration.

We don't need to install or configure tools like Webpack or Babel. They are preconfigured and hidden so that we can focus on the code. We can install easily just like any other node modules. Then it is just one command to start the React project.

```
create-react-app my-app
```

It includes everything we need to build a React app:

- React, JSX, ES6, and Flow syntax support.

- Language extras beyond ES6 like the object spread operator.
- Autoprefixed CSS, so you don't need `-webkit-` or other prefixes.
- A fast interactive unit test runner with built-in support for coverage reporting.
- A live development server that warns about common mistakes.
- A build script to bundle JS, CSS, and images for production, with hashes and sourcemaps.

Q10. What is Redux?

The basic idea of Redux is that the entire application state is kept in a single store. The store is simply a javascript object. The only way to change the state is by firing actions from your application and then writing reducers for these actions that modify the state. The entire state transition is kept inside reducers and should not have any side-effects.

Redux is based on the idea that there should be only a single source of truth for your application state, be it UI state like which tab is active or Data state like the user profile details.

```
{
  first_name: 'John',
  last_name: 'Doe',
  age: 28
}
```

All of these data is retained by redux in a closure that redux calls a store . It also provides us a recipe of creating the said store, namely `createStore(x)` .

The `createStore` function accepts another function, `x` as an argument. The passed in function is responsible for returning the state of the application at that point in time, which is then persisted in the store. This passed in function is known as the `reducer` .

This is a valid example reducer function:

```
export default function reducer(state={}, action) {
  return state;
}
```

This store can only be updated by dispatching an action. Our App dispatches an `action`, it is passed into `reducer`; the reducer returns a fresh instance of the `state`; the store notifies our App and it can begin it's re render as required.

Q11. What is Redux Thunk used for?

Redux thunk is middleware that allows us to write action creators that return a function instead of an action. The thunk can then be used to delay the dispatch of an action if a certain condition is met. This allows us to handle the asyncronous dispatching of actions. The inner function receives the store methods `dispatch` and `getState` as parameters.

To enable Redux Thunk, we need to use `applyMiddleware()` as below

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './reducers/index';

// Note: this API requires redux@>=3.1.0
const store = createStore(
  rootReducer,
  applyMiddleware(thunk)
);
```

Q12. What is `PureComponent`? When to use `PureComponent` over `Component`?

`PureComponent` is exactly the same as Component except that it handles the `shouldComponentUpdate` method for us. When `props` or `state` changes, `PureComponent` will do a shallow comparison on both `props` and `state`. `Component` on the other hand won't compare current props and state to next out of the box. Thus, the component will re-render by default whenever `shouldComponentUpdate` is called.

When comparing previous `props` and `state` to next, a shallow comparison will check that primitives have the same value (eg, 1 equals 1 or that true equals true) and that the references are the same between more complex javascript values like objects and arrays.

It is good to prefer `PureComponent` over `Component` whenever we never mutate our objects.

```
class MyComponent extends React.PureComponent {

  render() {
    return <div>Hello World!</div>
  }

}
```

Q13. How Virtual-DOM is more efficient than Dirty checking?

In React, each of our components have a state. This state is like an observable. Essentially, React knows when to re-render the scene because it is able to observe when this data changes. Dirty checking is slower than observables because we must poll the data at a regular interval and check all of the values in the data structure recursively. By comparison, setting a value on the state will signal to a listener that some state has changed, so React can simply listen for change events on the state and queue up re-rendering.

The virtual DOM is used for efficient re-rendering of the DOM. This isn't really related to dirty checking your data. We could re-render using a virtual DOM with or without dirty checking. In fact, the diff algorithm is a dirty checker itself.

We aim to re-render the virtual tree only when the state changes. So using an observable to check if the state has changed is an efficient way to prevent unnecessary re-renders, which would cause lots of unnecessary tree diffs. If nothing has changed, we do nothing.

Q14. Is `setState()` is async? Why is `setState()` in React Async instead of Sync?

`setState()` actions are asynchronous and are batched for performance gains. This is explained in documentation as below.

`setState()` does not immediately mutate this.state but creates a pending state transition. Accessing this.state after calling this method can potentially return the existing value. There is no guarantee of

synchronous operation of calls to setState and calls may be batched for performance gains.

This is because setState alters the state and causes rerendering. This can be an expensive operation and making it synchronous might leave the browser unresponsive. Thus the setState calls are asynchronous as well as batched for better UI experience and performance.

Q15. What is `render()` in React? And explain its purpose?

Each React component must have a `render()` mandatorily. It returns a single React element which is the representation of the native DOM component. If more than one HTML element needs to be rendered, then they must be grouped together inside one enclosing tag such as `<form>`, `<group>`, `<div>` etc. This function must be kept pure i.e., it must return the same result each time it is invoked.

Q16. What are controlled and uncontrolled components in React?

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

- A **Controlled Component** 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".
- A Uncontrolled Component 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.

In most (or all) cases we should use controlled components.

Q17. Explain the components of Redux.

Redux is composed of the following components:

- **Action**—Actions are payloads of information that send data from our application to our store. They are the only source of information for the store. We send them to the store using

`store.dispatch()` . Primarly, they are just an object describes what happened in our app.

- **Reducer**—Reducers specify how the application's state changes in response to actions sent to the store. Remember that actions only describe what happened, but don't describe how the application's state changes. So this place determines how state will change to an action.

- **Store**—The Store is the object that brings Action and Reducer together. The store has the following responsibilities: Holds application state; Allows access to state via `getState()` ; Allows state to be updated via `dispatch(action)` ; Registers listeners via `subscribe(listener)` ; Handles unregistering of listeners via the function returned by `subscribe(listener)` .

It's important to note that we'll only have a single store in a Redux application. When we want to split your data handling logic, we'll use reducer composition instead of many stores.

Q18. What is `React.cloneElement` ? And the difference with `this.props.children` ?

`React.cloneElement` clone and return a new React element using using the passed `element` as the starting point. The resulting element will have the original element's props with the new props merged in shallowly. New children will replace existing children. `key` and `ref` from the original element will be preserved.

`React.cloneElement` only works if our child is a single React element. For almost everything `{this.props.children}` is the better solution. Cloning is useful in some more advanced scenarios, where a parent send in an element and the child component needs to change some props on that element or add things like ref for accessing the actual DOM element.

Q19. What is the second argument that can optionally be passed to `setState` and what is its purpose?

A callback function which will be invoked when `setState` has finished and the component is re-rendered.

Since the `setState` is asynchronous, which is why it takes in a second callback function. With this function, we can do what we want immediately after state has been updated.

Q20. What is the difference between React Native and React?

React is a JavaScript library, supporting both front end web and being run on the server, for building user interfaces and web applications.

On the other hand, React Native is a mobile framework that compiles to native app components, allowing us to build native mobile applications (iOS, Android, and Windows) in JavaScript that allows us to use ReactJS to build our components, and implements ReactJS under the hood.

With React Native it is possible to mimic the behavior of the native app in JavaScript and at the end, we will get platform specific code as the output. We may even mix the native code with the JavaScript if we need to optimize our application further.

For more React JS Interview Questions and answer use our Android App:

https://play.google.com/store/apps/details?id=com.vigowebs.interviewquestions