React Interview Questions

January 3 2017. 10 min read.

This post is part of our React Fundamentals course. ${salesPitch}. 👈's a joke. Carry on.

For the record, asking someone these questions probably isn't the best way to

For the record, asking someone these questions probably isn't the best way to get a deep understanding of their experience with React. ***React Interview Questions*** *just seemed like a better title than* ***Things you may or may not need to know in React but you may find helpful none the less****.*

What happens when you call ***setState****?*

The first thing React will do when setState is called is merge the object you passed into setState into the current state of the component. This will kick off a process called reconciliation. The end goal of reconciliation is to, in the most efficient way possible, update the UI based on this new state. To do this, React will construct a new tree of React elements (which you can think of as an object representation of your UI). Once it has this tree, in order to figure out how the UI should change in response to the new state, React will diff this new tree against the previous element tree. By doing this, React will then know the exact changes which occurred, and by knowing exactly what changes occurred, will able to minimize its footprint on the UI by only making updates where absolutely necessary.

What's the difference between an ***Element*** *and a* ***Component*** *in React?*

*What's the difference between an **Element** and a **Component** in React?*

Simply put, a React element describes what you want to see on the screen. Not so simply put, a React element is an object representation of some UI.

A React component is a function or a class which optionally accepts input and returns a React element (typically via JSX which gets transpiled to a `createElement` invocation).

For more info, check out React Elements vs React Components

*When would you use a **Class Component** over a **Functional Component**?*

If your component has state or a lifecycle method(s), use a Class component. Otherwise, use a Functional component.

*What are **refs** in React and why are they important?*

Refs are an escape hatch which allow you to get direct access to a DOM element or an instance of a component. In order to use them you add a ref attribute to

Refs are an escape hatch which allow you to get direct access to a DOM element or an instance of a component. In order to use them you add a ref attribute to your component whose value is a callback function which will receive the underlying DOM element or the mounted instance of the component as its first argument.

```
class UnControlledForm extends Component {
  handleSubmit = () => {
    console.log("Input Value: ", this.input.value)
  }
  render () {
    return (
      <form onSubmit={this.handleSubmit}>
        <input
          type='text'
          ref={(input) => this.input = input} />
        <button type='submit'>Submit</button>
      </form>
    )
  }
}
```

Above notice that our input field has a ref attribute whose value is a function. That function receives the actual DOM element of input which we then put on the instance in order to have access to it inside of the *handleSubmit* function.

It's often misconstrued that you need to use a class component in order to use refs, but refs can also be used with functional components by leveraging closures in JavaScript.

```
function CustomForm ({handleSubmit}) {
  let inputElement
```

```
function CustomForm ({handleSubmit}) {
  let inputElement
  return (
    <form onSubmit={() => handleSubmit(inputElement.va
      <input
        type='text'
        ref={(input) => inputElement = input} />
      <button type='submit'>Submit</button>
    </form>
  )
}
```

*What are **keys** in React and why are they important?*

Keys are what help React keep track of what items have changed, been added, or been removed from a list.

```
render () {
  return (
    <ul>
      {this.state.todoItems.map(({task, uid}) => {
        return <li key={uid}>{task}</li>
      })}
    </ul>
  )
}
```

```
          return <li key={uid}>{task}</li>
        })}
      </ul>
    )
  }
```

It's important that each key be unique among siblings. We've talked a few times already about reconciliation and part of this reconciliation process is performing a diff of a new element tree with the most previous one. Keys make this process more efficient when dealing with lists because React can use the key on a child element to quickly know if an element is new or if it was just moved when comparing trees. And not only do keys make this process more efficient, but without keys, React can't know which local state corresponds to which item on move. So never neglect keys when mapping.

*If you created a React element like **Twitter** below, what would the component definition of **Twitter** look like?*

```
<Twitter username='tylermcginnis33'>
  {(user) => user === null
    ? <Loading />
    : <Badge info={user} />}
</Twitter>
```

```
import React, { Component, PropTypes } from 'react'
import fetchUser from 'twitter'
```

```
import React, { Component, PropTypes } from 'react'
import fetchUser from 'twitter'
// fetchUser take in a username returns a promise
// which will resolve with that username's data.

class Twitter extends Component {
  // finish this
}
```

If you're not familiar with the *render callback* pattern, this will look a little strange. In this pattern, a component receives a function as its child. Take notice of what's inside the opening and closing `<Twitter>` tags above. Instead of another component as you've probably seen before, the *Twitter* component's child is a function. What this means is that in the implementation of the *Twitter* component, we'll need to treat *props.children* as a function.

Here's how I went about solving it.

```
import React, { Component, PropTypes } from 'react'
import fetchUser from 'twitter'

class Twitter extends Component {
  state = {
    user: null,
  }
  static propTypes = {
    username: PropTypes.string.isRequired,
  }
  componentDidMount () {
    fetchUser(this.props.username)
      .then((user) => this.setState({user}))
  }
  render () {
    return this.props.children(this.state.user)
```

```
        .then((user) => this.setState({user}))
    }
    render () {
      return this.props.children(this.state.user)
    }
  }
```

Notice that, just as I mentioned above, I treat *props.children* as a function by invoking it and passing it the user.

What's great about this pattern is that we've decoupled our parent component from our child component. The parent component manages the state and the consumer of the parent component can decide in which way they'd like to apply the arguments they receive from the parent to their UI.

To demonstrate this, let's say in another file we want to render a *Profile* instead of a *Badge*, because we're using the render callback pattern, we can easily swap around the UI without changing our implementation of the parent (*Twitter*) component.

```
<Twitter username='tylermcginnis33'>
  {(user) => user === null
    ? <Loading />
    : <Profile info={user} />}
</Twitter>
```

*What is the difference between a **controlled** component and an **uncontrolled** component?*

*What is the difference between a **controlled** component and an **uncontrolled** component?*

A large part of React is this idea of having components control and manage their own state. What happens when we throw native HTML form elements (input, select, textarea, etc) into the mix? Should we have React be the "single source of truth" like we're used to doing with React or should we allow that form data to live in the DOM like we're used to typically doing with HTML form elements? These two questions are at the heart of controlled vs uncontrolled components.

A **controlled** component is a component where React is in *control* and is the single source of truth for the form data. As you can see below, *username* doesn't live in the DOM but instead lives in our component state. Whenever we want to update *username*, we call *setState* as we're used to.

```
class ControlledForm extends Component {
  state = {
    username: ''
  }
  updateUsername = (e) => {
    this.setState({
      username: e.target.value,
    })
  }
  handleSubmit = () => {}
  render () {
    return (
      <form onSubmit={this.handleSubmit}>
        <input
          type='text'
          value={this.state.username}
          onChange={this.updateUsername} />
        <button type='submit'>Submit</button>
      </form>
```

```
            type='text'
            value={this.state.username}
            onChange={this.updateUsername} />
          <button type='submit'>Submit</button>
        </form>
      )
    }
  }
```

An **uncontrolled** component is where your form data is handled by the DOM, instead of inside your React component.

You use *refs* to accomplish this.

```
class UnControlledForm extends Component {
  handleSubmit = () => {
    console.log("Input Value: ", this.input.value)
  }
  render () {
    return (
      <form onSubmit={this.handleSubmit}>
        <input
          type='text'
          ref={(input) => this.input = input} />
        <button type='submit'>Submit</button>
      </form>
    )
  }
}
```

Though uncontrolled components are typically easier to implement since you just grab the value from the DOM using refs, it's typically recommended that you favor controlled components over uncontrolled components. The main reasons for this are that controlled components support instant field validation, allow you to conditionally disable/enable buttons, enforce input formats, and

just grab the value from the DOM using refs, it's typically recommended that you favor controlled components over uncontrolled components. The main reasons for this are that controlled components support instant field validation, allow you to conditionally disable/enable buttons, enforce input formats, and are more "the React way".

In which lifecycle event do you make AJAX requests and why?

AJAX requests should go in the **componentDidMount** lifecycle event.

There are a few reasons for this,

- Fiber, the next implementation of React's reconciliation algorithm, will have the ability to start and stop rendering as needed for performance benefits. One of the trade-offs of this is that **componentWillMount**, the other lifecycle event where it might make sense to make an AJAX request, will be "non-deterministic". What this means is that React may start calling *componentWillMount* at various times whenever it feels like it needs to. This would obviously be a bad formula for AJAX requests.
- You can't guarantee the AJAX request won't resolve before the component mounts. If it did, that would mean that you'd be trying to setState on an unmounted component, which not only won't work, but React will yell at you for. Doing AJAX in componentDidMount will guarantee that there's a component to update.

*What does **shouldComponentUpdate** do and why is it important?*

Above we talked about reconciliation and what React does when setState is called. What **shouldComponentUpdate** does is it's a lifecycle method that allows us to opt out of this reconciliation process for certain components (and their child components). Why would we ever want to do this? As mentioned above, "The end goal of reconciliation is to, in the most efficient way possible, update the UI based on new state". If we know that a certain section of our UI isn't going to change, there's no reason to have React go through the trouble of trying to figure out if it should. By returning false from **shouldComponentUpdate**, React will assume that the current component, and all its child components, will stay the same as they currently are.

*How do you tell React to build in **Production** mode and what will that do?*

Typically you'd use Webpack's *DefinePlugin* method to set **NODE_ENV** to **production**. This will strip out things like propType validation and extra warnings. On top of that, it's also a good idea to minify your code because React uses Uglify's dead-code elimination to strip out development only code and comments, which will drastically reduce the size of your bundle.

comments, which will drastically reduce the size of your bundle.

Why would you use `React.Children.map(props.children, () => )` *instead of* `props.children.map(() => )`

It's not guaranteed that *props.children* will be an array.

Take this code for example,

```
<Parent>
  <h1>Welcome.</h1>
</Parent>
```

Inside of Parent if we were to try to map over children using `props.children.map` it would throw an error because `props.children` is an object, not an array.

React only makes `props.children` an array if there are more than one child elements, like this

```
<Parent>
  <h1>Welcome.</h1>
  <h2>props.children will now be an array</h2>
</Parent>
```

This is why you want to favor `React.Children.map` because its implemention takes into account that *props.children* may be an array or an

```
    </Parent>
```

This is why you want to favor `React.Children.map` because its implemention takes into account that *props.children* may be an array or an object.

Describe how events are handled in React.

In order to solve cross browser compatibility issues, your event handlers in React will be passed instances of *SyntheticEvent*, which is React's cross-browser wrapper around the browser's native event. These synthetic events have the same interface as native events you're used to, except they work identically across all browsers.

What's mildly interesting is that React doesn't actually attach events to the child nodes themselves. React will listen to all events at the top level using a single event listener. This is good for performance and it also means that React doesn't need to worry about keeping track of event listeners when updating the DOM.

*What is the difference between **createElement** and **cloneElement**?*

createElement is what JSX gets transpiled to and is what React uses to create React Elements (object representations of some UI). *cloneElement* is used in

createElement is what JSX gets transpiled to and is what React uses to create React Elements (object representations of some UI). *cloneElement* is used in order to clone an element and pass it new props. They nailed the naming on these two 🙂.

*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.

Something that's not spoken of a lot is that setState is asynchronous, which is why it takes in a second callback function. Typically it's best to use another lifecycle method rather than relying on this callback function, but it's good to know it exists.

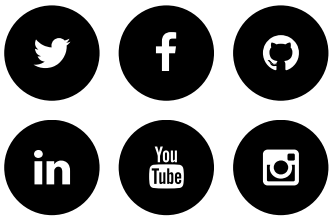
```
this.setState(
  { username: 'tylermcginnis33' },
  () => console.log('setState has finished and the com
)
```

What is wrong with this code?

```
this.setState((prevState, props) => {
  return {
    streak: prevState.streak + props.count
  }
})
```

Nothing is wrong with it 🙂. It's rarely used and not well known, but you can also pass a function to **setState** that receives the previous state and props and returns a new state, just as we're doing above. And not only is nothing wrong with it, but it's also actively recommended if you're setting state based on previous state.

Liked this post? Share it 🤦


You
Tube