

React Interview Questions and Answers

Q1) 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, 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 be able to minimize its footprint on the UI by only making updates where necessary.

Q2) 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 that optionally accepts input and returns a React element (typically via JSX which gets compiled to a `createElement` invocation).

Q3) 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 (or Hooks). Otherwise, use a Functional component.

Q4) 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. 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 {
  input = React.createRef()
  handleSubmit = () => {
    console.log("Input Value: ", this.input.current.value)
  }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <input
          type='text'
          ref={this.input} />
        <button type='submit'>Submit</button>
      </form>
    )
  }
}
```

Q5) 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.

```
function List({ todos }) {
  return (
```

```

      <ul>
        {todos.map(({ task, id }) => <li key={id}>{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 the 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.

Q6) If you created a React element like Twitter below, what would the component definition of Twitter look like?

```

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

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

Q7) 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>
      )
    }
  }
}

```

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 {
  input = React.createRef()
  handleSubmit = () => {
    console.log("Input Value: ", this.input.current.value)
  }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <input
          type='text'
          ref={this.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 are more "the React way".

Q8) In which lifecycle method do you make AJAX requests with a Class component?

AJAX requests should go in the `componentDidMount` lifecycle method.

The reason for this is because 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.

Q9) 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.

Note, you probably don't ever need to use shouldComponentUpdate. React is typically fast enough by default.

Q10) How do you tell React to build in Production mode and what will that do?

You set process.env.NODE_ENV to production. When React in production mode, it'll strip out any extra development features like warnings.

Q11) 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>
```


You want to favor `React.Children.map` because its implementation takes into account that `props.children` may be an array or an object.

Q12) 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.

Q13) What is the difference between `createElement` and `cloneElement`?

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

Q14) 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: 'tylermcginnis' },  
  () => console.log('setState has finished and the component has  
re-rendered.')  
)
```

Q15) 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 the previous state.

