*1*: **How does React *work*?**

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.

*Q2*: **What is *Context API* in ReactJS?**

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 “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} />;  }  } |
| --- |

*3*: **What are props in React?**

**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 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) 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; |
| --- |

*4*: **What is the use of refs?**

**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 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} />; }  } |
| --- |

*5*: **What are the *advantages* of ReactJS?**

Below are the advantages of ReactJS:

1. Increases the application’s performance with Virtual DOM
2. JSX makes code is easy to read and write
3. It renders both on 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.

*6*: **What are React Hooks?** Related To: [React Hooks](https://www.fullstack.cafe/interview-questions/react-hooks)

**Hooks** are a new addition in 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.

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

For example:

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

*8*: **What is React?**

React is an open-source JavaScript library created by Facebook for building complex, interactive UIs in web and mobile applications. React’s core purpose is to build UI components; it is often referred to as just the “V” (View) in an “MVC” architecture.

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

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

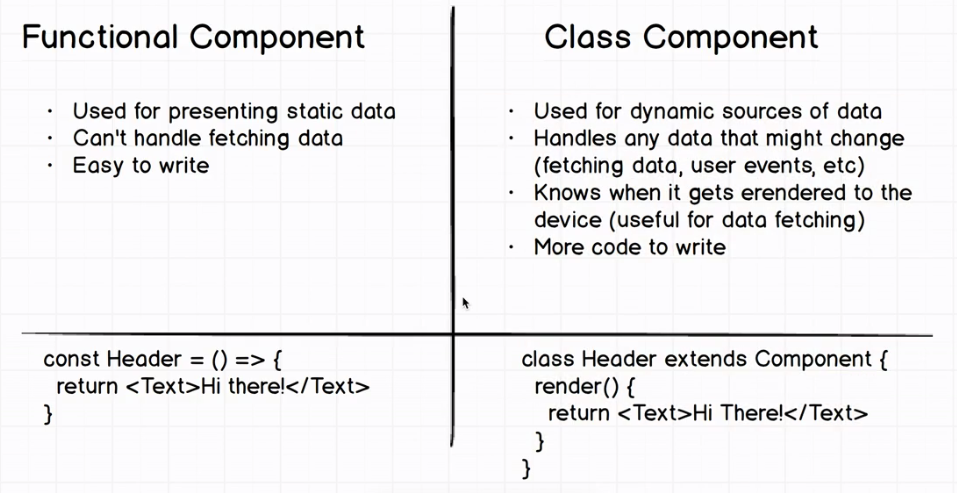
*10*: **What are the differences between a *Class component* and *Functional component*?**

**Class Components**

* Class-based Components uses ES6 class syntax. It can make use of the lifecycle methods.
* Class components extend from React.Component.
* In 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 comparing to class-based functions.
* Functional Components mainly focuses on the UI of the application, not on the behavior.
* To be more precise these are basically render function in the class component.
* Functional Components can have state and mimic lifecycle events using Reach Hooks



*11*: **What are the *advantages* of using React?**

* It is easy to know how a component is rendered, you just need to look at the render function.
* JSX makes it easy to read the code of your components. It is also really easy to see the layout, or how components are plugged/combined with each other.
* You can render React on the server-side. This enables improves SEO and performance.
* It is easy to test.
* You can use React with any framework (Backbone.js, Angular.js) as it is only a view layer.

*12*: **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. 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.

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

* **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 UI state, as opposed to 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.

*Q14***: What are refs used for in React?**

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  return (  <form onSubmit={() => handleSubmit(inputElement.value)}>  <input  type='text'  ref={(input) => inputElement = input} />  <button type='submit'>Submit</button>  </form>  )  } |
| --- |

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

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".
* A [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> |
| --- |

In most (or all) cases [you should use controlled components](https://goshakkk.name/controlled-vs-uncontrolled-inputs-react/#conclusion).

*16*: **What are *Controlled components* in ReactJS?**

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".

| // Controlled:  <input type="text" value={value} onChange={handleChange} /> |
| --- |

*17*: **What is state in React?**

**State** of a component is an object that holds some information that may change over the lifetime of the component. We should always try to make our state as simple as possible and minimize the number of stateful components.

| class User extends React.Component {  constructor(props) {  super(props);  this.state = {  message: "Welcome to React world",  }  }  render() {  return (  <div>  <h1>{this.state.message}</h1>  </div>  );  }  } |
| --- |

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

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

*19*: **What are *Fragments* in React?**

It's common pattern in React which is used for a component to *return multiple elements*. **Fragments** let you group a list of children without adding extra nodes to the DOM.

| render() {  return (  <React.Fragment>  <ChildA />  <ChildB />  <ChildC />  </React.Fragment>  );  } |
| --- |

There is also a **shorter syntax**:

| render() {  return (  <>  <ChildA />  <ChildB />  <ChildC />  </>  );  } |
| --- |

*20***: When rendering a list what is a key and what is it's purpose?**

*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.

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

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. It is not recommend to use indexes for keys if the items can reorder, as that would be slow.

*21*: **How to create refs in React?**

**Refs** are created using React.createRef() method and attached to React elements via the ref attribute. In order to use refs throughout the component, just assign the ref to the instance property with in constructor.

| class MyComponent extends React.Component {  constructor(props) {  super(props);  this.myRef = React.createRef();  }  render() {  return <div ref={this.myRef} />;  }  } |
| --- |

And:

| class UserForm extends Component {  handleSubmit = () => {  console.log("Input Value is: ", this.input.value)  }  render () {  return (  <form onSubmit={this.handleSubmit}>  <input  type='text'  ref={(input) => this.input = input} /> // Access DOM input in handle submit  <button type='submit'>Submit</button>  </form>  )  }  } |
| --- |

We can also use it in functional components with the help of closures.

***22*: What is useState() in React?**

**Problem**

Explain what is the use of useState(0) there:

| ...  const [count, setCounter] = useState(0);  const [moreStuff, setMoreStuff] = useState(...);  ...  const setCount = () => {  setCounter(count + 1);  setMoreStuff(...);  ...  }; |
| --- |

**Answer**

**useState** is one of build-in react hooks. useState(0) returns a tuple where the first parameter count is the current state of the counter and setCounter is the method that will allow us to update the counter's state.

We can use the setCounter method to update the state of count anywhere - In this case we are using it inside of the setCount function where we can do more things; the idea with hooks is that we are able to keep our code more functional and avoid class based components if not desired/needed.

*23*: **What are *Stateful* components in React?**

If the behaviour of a component is dependent on the state of the component then it can be termed as *stateful component*. These Stateful components are always class components and have a state that gets initialized in the constructor.

| class App extends Component {  constructor(props) {  super(props);  this.state = { count: 0 };  }  render() {  // omitted for brevity  }  } |
| --- |

*24*: **What is *JSX*?**

JSX is a syntax notation for **JavaScript XML** (XML-like syntax extension to ECMAScript). It stands for JavaScript XML. It provides expressiveness of JavaScript along with HTML like template syntax. For example, the below text inside h1 tag return as javascript function to the render function,

| render(){  return(  <div>  <h1> Welcome to React world!!</h1>  </div>  );  } |
| --- |

*25*: **What are the *limitations* of React?**

Below are the list of limitations:

1. React is just a view library, not a full-blown framework
2. There is a learning curve for beginners who are new to web development.
3. Integrating React.js into a traditional MVC framework requires some additional configuration
4. The code complexity increases with inline templating and JSX.
5. Too many smaller components leading to over-engineering or boilerplate

*26*: **What are *Stateless components* in React?**

If the behaviour 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, and test, and
* you can avoid the this keyword altogether.

*27*: **How is React different from AngularJS (1.x)?**

For example, AngularJS (1.x) approaches building an application by extending HTML markup and injecting various constructs (e.g. Directives, Controllers, Services) at runtime. As a result, AngularJS is very opinionated about the greater architecture of your application — these abstractions are certainly useful in some cases, but they come at the cost of flexibility.

By contrast, React focuses exclusively on the creation of components, and has few (if any) opinions about an application’s architecture. This allows a developer an incredible amount of flexibility in choosing the architecture they deem “best” — though it also places the responsibility of choosing (or building) those parts on the developer.

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

Both **props** and **state** are plain JavaScript objects. While both of them hold information that influences the output of render, they are different in their functionality with respect to component. i.e,

* **Props** get passed to the component similar to function parameters
* **State** is managed within the component similar to variables declared within a function.

*29*: **What are *two types* of components in ReactJS?**

There are two possible ways to create ReactJS Components.

1. **Functional components:** This is the simplest way to create ReactJS components. It accepts props as an Object and returns ReactJS elements. We call it as “functional” because those are pure JavaScript functions.

| function Greeting(props) {  return <h1> Hello, {props.message}</h1>  } |
| --- |

1. **Class components:** You can also use Es6 class to define component. The above functional component can be written as below,

| class Greeting extends React.Component {  render() {  return <h1>Hello, {this.props.message}</h1>;  }  } |
| --- |

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

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 lifecycle method rather this callback function.

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

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

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.

*32*: **What are advantages of using React Hooks?** Related To: [React Hooks](https://www.fullstack.cafe/interview-questions/react-hooks)

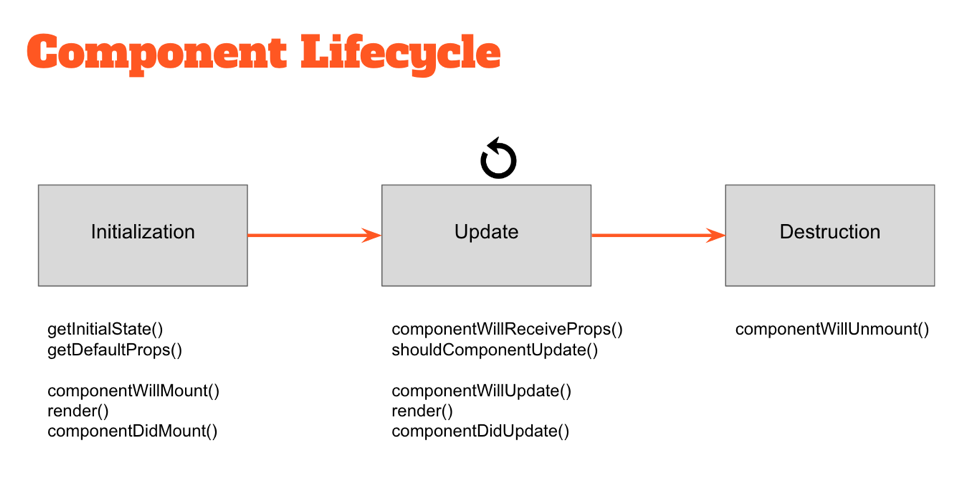
Primarily, hooks in general enable the extraction and reuse of stateful logic that is common across multiple components without the burden of higher order components or render props. Hooks allow to easily manipulate the state of our functional component without needing to convert them into class components.

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 .

*33*: **What happens during the lifecycle of a React component?**

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

1. Initialization
2. State/Property Updates
3. Destruction



*34*: **What is the difference between *Component* and *Container* in Redux?** Related To: [Redux](https://www.fullstack.cafe/interview-questions/redux)

* **Component** is part of the React API. A Component is a class or function that describes part of a React UI.
* **Container** is an informal term for a React component that is connected to a redux store. Containers receive Redux state updates and dispatch actions, and they usually don't render DOM elements; they delegate rendering to presentational child components.

*35*: **What are *inline conditional expressions* in ReactJS?**

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>  )  } |
| --- |

*36*: **What is *Reconciliation* in ReactJS?**

When a component’s props or state change, React decides whether an actual DOM update is necessary by comparing the newly returned element with the previously rendered one. When they are not equal, React will update the DOM. This process is called **reconciliation**.

*Q37***: What is the purpose of using super constructor with props argument in React?**

A child class constructor cannot make use of this reference until super() method has been called. The same applies for ES6 sub-classes as well. The main reason of passing props parameter to super() call is to access this.props in your child constructors.

Passing props:

| class MyComponent extends React.Component {  constructor(props) {  super(props);  console.log(this.props); // Prints { name: 'sudheer',age: 30 }  }  } |
| --- |

Not passing props:

| class MyComponent extends React.Component {  constructor(props) {  super();  console.log(this.props); // Prints undefined  // But Props parameter is still available  console.log(props); // Prints { name: 'sudheer',age: 30 }  }  render() {  // No difference outside constructor  console.log(this.props) // Prints { name: 'sudheer',age: 30 }  }  } |
| --- |

The above code snippets reveals that this.props behavior is different only with in the constructor. It would be same outside the constructor.

*38*: **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.

3*9*: **What is the difference between *Element* and *Component* in ReactJS?**

An **element** is a plain object describing what you want to appear on the screen in terms of the DOM nodes or other components. Elements can contain other elements in their props. Creating a React element is cheap. Once an element is created, it is never mutated. The object representation of React element would be as follows,

| const element = React.createElement(  'div',  {id: 'login-btn'},  'Login'  )  The above createElement returns as object as below,  {  type: 'div',  props: {  children: 'Login',  id: 'login-btn'  }  } |
| --- |

And finally it renders to the DOM using ReactDOM.render as below,

| <div id='login-btn'>Login</div> |
| --- |

Whereas a **component** can be declared in several different ways. It can be a class with a render() method. Alternatively, in simple cases, it can be defined as a function. In either case, it takes props as an input, and returns an element tree as the output. JSX transpiled as createElement at the end.

| function Button ({ onLogin }) {  return React.createElement(  'div',  {id: 'login-btn', onClick: onLogin},  'Login'  )  } |
| --- |

*40*: **What are *Higher-Order Components* (HOC) in React?**

A higher-order component **(HOC)** is a function that takes a component and returns a new component. Basically, it’s a pattern that is derived from React’s compositional nature We call them as **“pure’ components”** because they can accept any dynamically provided child component but they won’t modify or copy any behavior from their input components.

| const EnhancedComponent = higherOrderComponent(WrappedComponent); |
| --- |

HOC can be used for many use cases as below,

1. Code reuse, logic and bootstrap abstraction
2. Render High jacking
3. State abstraction and manipulation
4. Props manipulation

*41***: How to call loading function with React useEffect only once?**

If you only want to run the function given to useEffect after the *initial render*, you can give it an *empty array* [] as the second argument.

For example:

| function MyComponent(){  useEffect(() => {  loadDataOnlyOnce();  }, []);  return <div> { /\*...\*/} </div>;  } |
| --- |

*42*: **How to access DOM elements in React?**

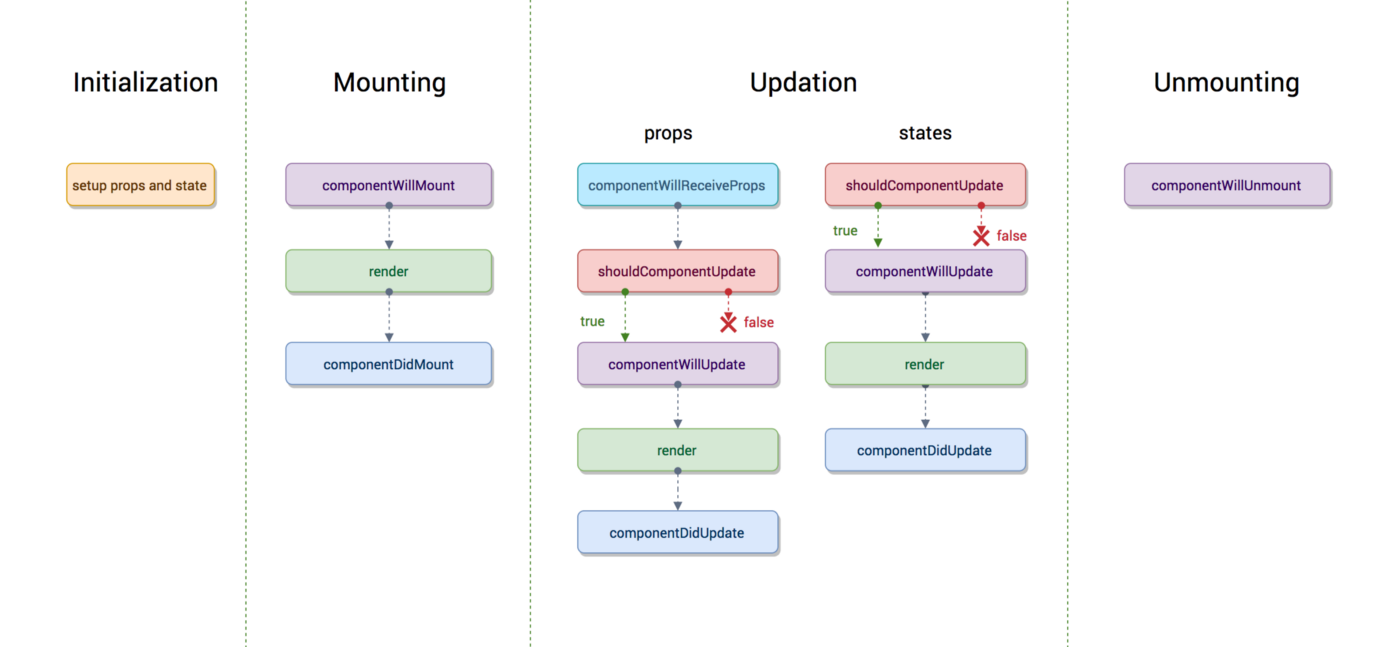
One of the useful application of the useRef() hook is to access DOM elements. This is performed in 3 steps:

1. Define the reference to access the element const elementRef = useRef();
2. Assign the reference to ref attribute of the element: <div ref={elementRef}></div>;
3. After mounting, elementRef.current points to the DOM element.

| import { useRef, useEffect } from 'react';  function AccessingElement() {  const elementRef = useRef();  useEffect(() => {  const divElement = elementRef.current;  console.log(divElement); // logs <div>I'm an element</div>  }, []);  return (  <div ref={elementRef}>  I'm an element  </div>  );  } |
| --- |

*43*: **Name the different *lifecycle* methods for a class components**

* componentWillMount- this is most commonly used for App configuration in your root component.
* componentDidMount - here you want to do all the setup you couldn’t do without a DOM, and start getting all the data you need. Also if you want to set up eventListeners etc. this lifecycle hook is a good place to do that.
* componentWillReceiveProps - this lifecyclye acts on particular prop changes to trigger state transitions.
* shouldComponentUpdate - if you’re worried about wasted renders shouldComponentUpdate is a great place to improve performance as it allows you to prevent a rerender if component receives new prop. shouldComponentUpdate should always return a boolean and based on what this is will determine if the component is rerendered or not.
* componentWillUpdate - rarely used. It can be used instead of componentWillReceiveProps on a component that also has shouldComponentUpdate (but no access to previous props).
* componentDidUpdate - also commonly used to update the DOM in response to prop or state changes.
* componentWillUnmount - here you can cancel any outgoing network requests, or remove all event listeners associated with the component.



*44*: **What is {this.props.children} and when you should use it?**

You can use props.children on components that represent ‘generic boxes’ and that don’t know their children ahead of time. It is used to display whatever you include between the opening and closing tags when invoking a component.

| const Picture = (props) => {  return (  <div>  <img src={props.src}/>  {props.children}  </div>  )  } |
| --- |

*45*: **How would you prevent a component from *rendering* in React?**

Returning null from a component's render method does not affect the firing of the component's lifecycle methods.

*46***: What's the typical pattern for *rendering a list of components* from an array in React?**

Call map on an array with an arrow function that executes for each array element, possibly outputting a React component for each.

| const numbers = [1, 2, 3, 4, 5];  const listItems = numbers.map((number) =>  <li key={number.toString()}>  {number}  </li>  ); |
| --- |

*47*: **What are *Pure Components*?**

**PureComponent** is exactly the same as **Component** except that it handles the shouldComponentUpdate method for you.

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.

*48*: **What's the typical flow of data like in a React + Redux app?**

Callback from UI component dispatches an action with a payload, which then is intercepted in a reducer, possibly producing a new application state, which is then propagated down through the tree of components in the application from the Redux store.

*49*: **What are some *limitations* of things you shouldn't do in the component's *render* method in React?**

You cannot modify the component's state (with setState), nor interact with the browser (do that in componentDidMount). Render should be a **pure function**.

*50*: **How to *bind* methods or event handlers in JSX callbacks?**

There are 3 possible ways to achieve,

**1 Binding in Constructor:** In JavaScript classes, the methods are not bound by default. The same thing applies for ReactJS event handlers defined as class methods. Normally we bind them in constructor as follows,

| constructor(props) {  super(props);  this.handleClick = this.handleClick.bind(this);  }  handleClick() {  // Perform some logic  } |
| --- |

**2 Public class fields syntax:** If you don’t like to use bind approach then public class fields syntax can be used to correctly bind callbacks

| handleClick = () => {  console.log('this is:', this);  }  <button onClick={this.handleClick}>  Click me  </button> |
| --- |

**3 Arrow functions in callbacks:** You can use arrow functions directly in the callbacks as below

| <button onClick={(e) => this.handleClick(e)}>  Click me  </button> |
| --- |

*51*: **What is *prop drilling* and how can you avoid it?**

When building a React application, there is often the need for a deeply nested component to use data provided by another component that is much higher in the hierarchy. The simplest approach is to simply pass a prop from each component to the next in the hierarchy from the source component to the deeply nested component. This is called **prop drilling**.

The primary disadvantage of prop drilling is that components that should not otherwise be aware of the data become unnecessarily complicated and are harder to maintain.

To avoid prop drilling, a common approach is to use React context. This allows a Provider component that supplies data to be defined, and allows nested components to consume context data via either a Consumer component or a useContext hook.

*52*: **What is the point of shouldComponentUpdate() method?**

It's used for performance reasons, for example if the implementor of a component knows for sure that a particular property change does not necessitate a re-render, they could return false from this method and skip the re-render.

*53***: What are forward refs?**

Ref forwarding is a feature that lets some components take a ref they receive, and pass it further down to a child.

| const ButtonElement = React.forwardRef((props, ref) => (  <button ref={ref} className="CustomButton">  {props.children}  </button>  ));  // Create ref to the DOM button:  const ref = React.createRef();  <ButtonElement ref={ref}>Forward Ref</ButtonElement>; |
| --- |

*54***: What do these three dots (...) in React do?**

**Problem**

What does the ... do in this React (using JSX) code and what is it called?

| <Modal {...this.props} title='Modal heading' animation={false}/> |
| --- |

**Answer**

That's property spread notation. It was added in ES2018 (spread for arrays/iterables was earlier, ES2015).

For instance, if this.props contained a: 1 and b: 2, then

| <Modal {...this.props} title='Modal heading' animation={false}> |
| --- |

would be the same as:

| <Modal a={this.props.a} b={this.props.b} title='Modal heading' animation={false}> |
| --- |

Spread notation is handy not only for that use case, but for creating a new object with most (or all) of the properties of an existing object — which comes up a lot when you're updating state, since you can't modify state directly:

| this.setState(prevState => {  return {foo: {...prevState.foo, a: "updated"}};  }); |
| --- |

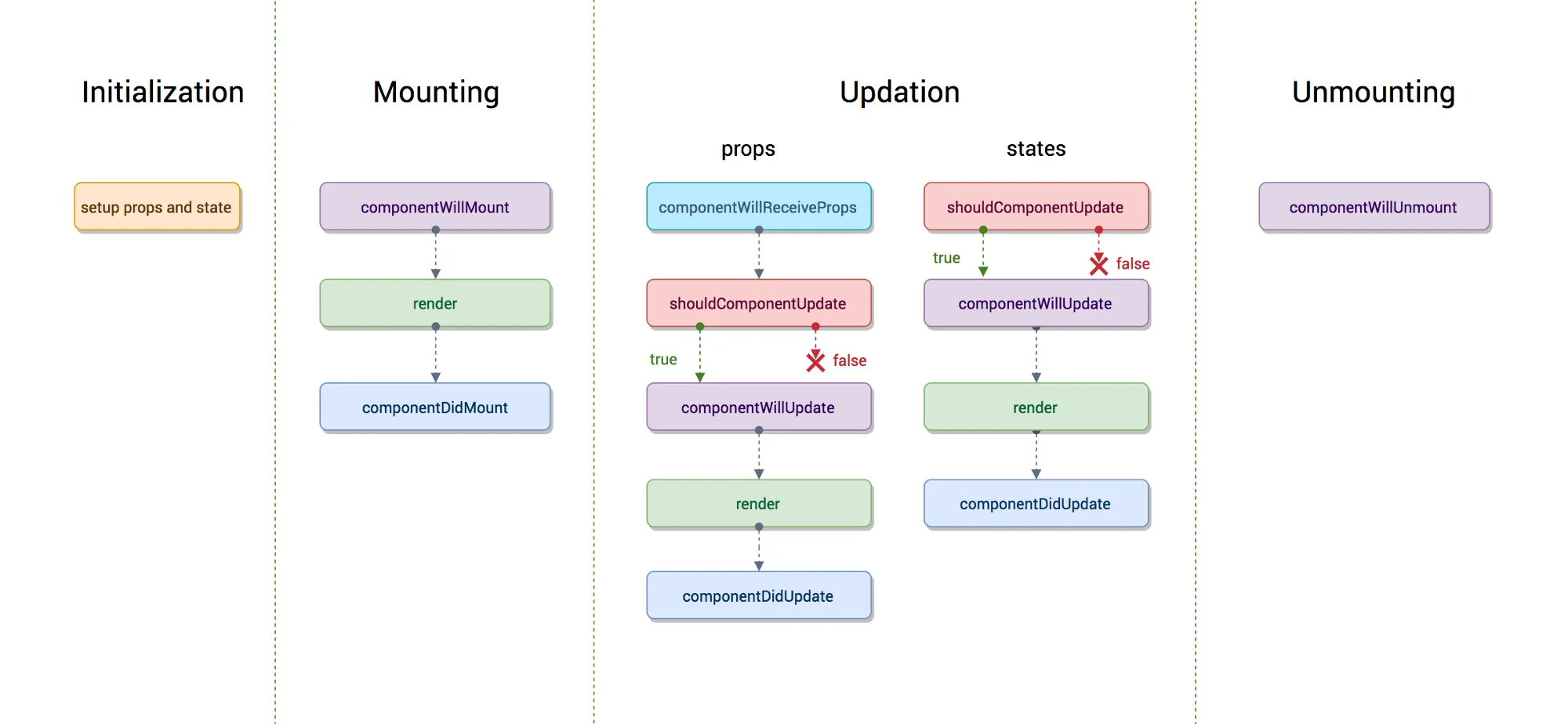
*55*: **What are the *lifecycle* methods of ReactJS class components?**

* componentWillMount: Executed before rendering and is used for App level configuration in your root component.
* componentDidMount: Executed after first rendering and here all AJAX requests, DOM or state updates, and set up eventListeners should occur.
* componentWillReceiveProps: Executed when particular prop updates to trigger state transitions.
* shouldComponentUpdate: Determines if the component will be updated or not. By default it returns true. If you are sure that the component doesn't need to render after state or props are updated, you can return false value. It is a great place to improve performance as it allows you to prevent a rerender if component receives new prop.
* componentWillUpdate: Executed before re-rendering the component when there are pros & state changes confirmed by shouldComponentUpdate which returns true.
* componentDidUpdate: Mostly it is used to update the DOM in response to prop or state changes.
* componentWillUnmount: It will be used to cancel any outgoing network requests, or remove all event listeners associated with the component.

**What are the different *phases* of ReactJS component *lifecycle*?**

There are four different phases of React component’s lifecycle:

1. **Initialization:** In this phase react component prepares setting up the initial state and default props.
2. **Mounting:** The react component is ready to mount in the browser DOM. This phase covers **componentWillMount** and **componentDidMount** lifecycle methods.
3. **Updating:** In this phase, the component get updated in two ways, sending the new props and updating the state. This phase covers **shouldComponentUpdate, componentWillUpdate and componentDidUpdate** lifecycle methods.
4. **Unmounting:** In this last phase, the component is not needed and get unmounted from the browser DOM. This phase include **componentWillUnmount** lifecycle method.



*57*: **What is Key and benefit of using it in lists?**

A **key** is a special string attribute you need to include when creating lists of elements. Keys help React identify which items have changed, are added, or are removed.

For example, most often we use IDs from your data as keys

| const todoItems = todos.map((todo) =>  <li key={todo.id}>  {todo.text}  </li>  ); |
| --- |

When you don’t have stable IDs for rendered items, you may use the item index as a key as a last resort:

| const todoItems = todos.map((todo, index) =>  <li key={index}>  {todo.text}  </li>  ); |
| --- |

**Note:**

1. 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
2. If you extract list item as separate component then apply keys on list component instead li tag.

There will be a warning in the console if the key is not present on list items.

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

* **Elements** are the fundamental building blocks of React, and describe what you want to see on the screen. They are just simple JS objects with props, key, ref, and type properties, whereas
* **Components** have a render method and optionally accept inputs.

*59*: **What is the difference between *ShadowDOM* and *VirtualDOM*?**

**Virtual DOM**

Virtual DOM is about avoiding unnecessary changes to the DOM, which are expensive performance-wise, because changes to the DOM usually cause re-rendering of the page. Virtual DOM also allows to collect several changes to be applied at once, so not every single change causes a re-render, but instead re-rendering only happens once after a set of changes was applied to the DOM.

**Shadow DOM**

Shadow dom is mostly about encapsulation of the implementation. A single custom element can implement more-or-less complex logic combined with more-or-less complex DOM. An entire web application of arbitrary complexity can be added to a page by an import and <body><my-app></my-app> but also simpler reusable and composable components can be implemented as custom elements where the internal representation is hidden in the shadow DOM like

<date-picker></date-picker>.

*60*: **Why do class methods need to be *bound* to a class instance?**

In JavaScript, the value of this changes depending on the current context. Within React class component methods, developers normally expect this to refer to the current instance of a component, so it is necessary to *bind* these methods to the instance. Normally this is done in the constructor—for

| class SubmitButton extends React.Component {  constructor(props) {  super(props);  this.state = {  isFormSubmitted: false  };  this.handleSubmit = this.handleSubmit.bind(this);  }  handleSubmit() {  this.setState({  isFormSubmitted: true  });  }  render() {  return (  <button onClick={this.handleSubmit}>Submit</button>  )  }  } |
| --- |

*61***: What are *Stateless* components in React?**

Stateless components (a flavor of “reusable” components) are nothing more than pure functions that render DOM based solely on the properties provided to them.

| const StatelessCmp = props => {  return (  <div className="my-stateless-component">  {props.name}: {props.birthday}  </div>  );  };  // ---  ReactDOM.render(  <StatelessCmp name="Art" birthday="10/01/1980" />,  document.getElementById('main')  ); |
| --- |

This component has no need for any internal state — let alone a constructor or lifecycle handlers. The output of the component is purely a function of the properties provided to it.

*Q62*: **What is children prop?**

Children is a prop (this.prop.children) that allow you to pass components as data to other components, just like any other prop you use.

There are a number of methods available in the React API to work with this prop. These include:

* React.Children.map,
* React.Children.forEach,
* React.Children.count,
* React.Children.only,
* React.Children.toArray.

A simple usage of children prop looks as below,

| var MyDiv = React.createClass({  render: function() {  return <div>{this.props.children}</div>;  }  });  ReactDOM.render(  <MyDiv>  <span>Hello</span>  <span>World</span>  </MyDiv>,  node  ); |
| --- |

*63***: Why React uses className over class attribute?**

class is a keyword in javascript and JSX is an extension of javascript. That's the principal reason why React uses className instead of class.

| render() {  return <span className="menu navigation-menu">Menu</span>  } |
| --- |

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

What shouldComponentUpdate does is it’s a lifecycle method that allows us to opt out of *setState* reconciliation process for certain components (and their child components). 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.

*65*: **What is *Lifting State Up* in ReactJS?**

When several components need to share the same changing data then it is recommended to lifting the shared state up to their closest common ancestor. For example, if two child components sharing the same data from its parent then move the state to parent instead of maintaining the local state inn both child components.

*66***: Why we should not update state directly?**

If you try to update state directly then it won’t re-render the component.

//Wrong

| This.state.message =”Hello world”; |
| --- |

Instead use setState() method. It schedules an update to a component’s state object. When state changes, the component responds by re-rendering

//Correct

| this.setState({message: ‘Hello World’}); |
| --- |

Note: The only place you can assign the state is constructor.

*67*: **What's the difference between useRef and createRef?**

The difference is:

* createRef will always create a new ref. In a class-based component, you would typically put the ref in an instance property during construction (e.g. this.input = createRef()). You don't have this option in a function component.
* useRef takes care of returning the same ref each time as on the initial rendering.

*68*: **What is StrictMode in React?**

React's StrictMode is sort of a helper component that will help you write better react components, you can wrap a set of components with <StrictMode /> and it'll basically:

* Verify that the components inside are following some of the recommended practices and warn you if not in the console.
* Verify the deprecated methods are not being used, and if they're used strict mode will warn you in the console.
* Help you prevent some side effects by identifying potential risks.

*69*: **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 order to clone an element and pass it new props. They nailed the naming on these two.

*70***: What is the significance of keys in ReactJS?**

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:

Example:

| const numbers = [1, 2, 3, 4, 5];  const listItems = numbers.map((number) =>  <li key={number.toString()}>  {number}  </li>  ); |
| --- |

When React reconciles the keyed children, it will ensure that any child with key will be reordered (instead of clobbered) or destroyed (instead of reused).

*71***: What would be the common mistake of function being called every time the component renders?**

You need to make sure that function is not being called while passing the function as a parameter.

| render() {  // Wrong way: handleClick is called instead of passed as a reference!  return <button onClick={this.handleClick()}>Click Me</button>  } |
| --- |

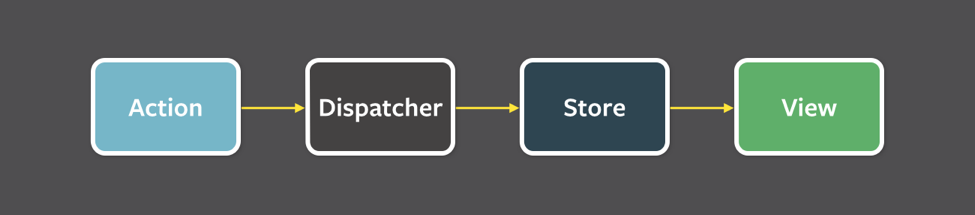
Instead, pass the function itself without parenthesis:

| render() {  // Correct way: handleClick is passed as a reference!  return <button onClick={this.handleClick}>Click Me</button>  } |
| --- |

*72*: **Are you familiar with Flux in the context of React?**

**Flux** is an architectural pattern that enforces unidirectional data flow — its core purpose is to control derived data so that multiple components can interact with that data without risking pollution.

The Flux pattern is generic; it’s not specific to React applications, nor is it required to build a React app. However, Flux is commonly used by React developers because React components are declarative — the rendered UI (View) is simply a function of state (Store data).



In the Flux pattern, the *Store* is the central authority for all data; any mutations to the data must occur within the store. Changes to the Store data are subsequently broadcast to subscribing Views via events. Views then update themselves based on the new state of received data.

To request changes to any Store data, *Actions* may be fired. These Actions are controlled by a central *Dispatcher*; Actions may not occur simultaneously, ensuring that a Store only mutates data once per Action.

The strict unidirectional flow of this Flux pattern enforces data stability, reducing data-related runtime errors throughout an application.

*73***: What is the difference between HTML and React *event handling*?**

**Below are the few differences between HTML and React event handling:**

1 In HTML, the event name should be in lowercase.

| <button onclick="activateLasers()"> |
| --- |

Whereas in ReactJS it follows camelCase convention,

| <button onClick={activateLasers}> |
| --- |

2 In HTML, you can return false to prevent default behavior,

| <a href="#" onclick="console.log('The link was clicked.'); return false"/> |
| --- |

3 Whereas in ReactJS you must call preventDefault explicitly,

| function handleClick(e) {  e.preventDefault();  console.log('The link was clicked.');  } |
| --- |

*74***: What are *Error Boundaries* in ReactJS?**

Error boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed.

A class component becomes an error boundary if it defines a new lifecycle method called componentDidCatch(error, info)

| class ErrorBoundary extends React.Component {  constructor(props) {  super(props);  this.state = { hasError: false };  }  componentDidCatch(error, info) {  // Display fallback UI  this.setState({ hasError: true });  // You can also log the error to an error reporting service  logErrorToMyService(error, info);  }  render() {  if (this.state.hasError) {  // You can render any custom fallback UI  return <h1>Something went wrong.</h1>;  }  return this.props.children;  }  } |
| --- |

After that use it as a regular component

| <ErrorBoundary>  <MyWidget />  </ErrorBoundary> |
| --- |

*75*: **What are *Uncontrolled components*?**

The **Uncontrolled Component** are the 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 For example, in the below UserProfile component, the name input accessed using ref as below,

| class UserProfile extends React.Component {  constructor(props) {  super(props);  this.handleSubmit = this.handleSubmit.bind(this);  this.input = React.createRef();  }  handleSubmit(event) {  alert('A name was submitted: ' + this.input.current.value);  event.preventDefault();  }  render() {  return (  <form onSubmit={this.handleSubmit}>  <label>  Name:  <input type="text" ref={this.input} />  </label>  <input type="submit" value="Submit" />  </form>  );  }  } |
| --- |

In most cases, it is recommend using controlled components to implement forms.

*76***: What's wrong with using *Context* in React?**

* 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**.
* If you only want to avoid passing some props through many levels, component composition is often a simpler solution than context.

For example, consider a Page component that passes a user and avatarSize prop several levels down so that deeply nested Link and Avatar components can read it:

| <Page user={user} avatarSize={avatarSize} />  // ... which renders ...  <PageLayout user={user} avatarSize={avatarSize} />  // ... which renders ...  <NavigationBar user={user} avatarSize={avatarSize} />  // ... which renders ...  <Link href={user.permalink}>  <Avatar user={user} size={avatarSize} />  </Link> |
| --- |

It might feel redundant to pass down the user and avatarSize props through many levels if in the end only the Avatar component really needs it. It’s also annoying that whenever the Avatar component needs more props from the top, you have to add them at all the intermediate levels too.

*77*: **What is *Components Composition* in React?**

Sometimes we think about components as being “special cases” of other components. For example, we might say that a WelcomeDialog is a special case of Dialog.

In React, this is also achieved by **components composition**, where a more “specific” component renders a more “generic” one and configures it with props:

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

*78***: What does *Batching* mean in ReactJS? Related To:** [**React Hooks**](https://www.fullstack.cafe/interview-questions/react-hooks)

Batching is nothing but grouping React multiple state updates together into a single render state to achieve better computational performance. Until React 18, we only *batched* updates during the *React event handlers*. Updates inside of promises, setTimeout, *native event handlers***, or any other event were not** *batched* **in React by default.**

| function App() {  const [count, setCount] = useState(0);  const [flag, setFlag] = useState(false);  function handleClick() {  setCount(c => c+1); // Does not re-render yet  setFlag(f => !f); // Does not re-render yet  // React will only re-render once at the end (that's batching!)  }  return (  <div>  <button onClick={handleClick}>Next</button>  <h1 style={{color: flag ? "blue": "black"}}>{count}</h1>  </div>  )  } |
| --- |

*79*: **What are the advantages of *Batching* in ReactJS?**

* **Batching** is great for *performance* because it avoids unnecessary re-renders.
* **Batching** also *prevents* your component from *rendering* half-finished states where only one state variable was updated, which may cause bugs.
* Another reason to use **batching** is when the web application grows, the number of nested components will increase. Therefore, if a parent component executes an *unbatched state updated*, the *entire component tree will be re-rendered per state update* that is expensive.

*80*: **Which *lifecycle* methods of class component is replaced by useEffect in functional component?**

The lifecyce methods replaced by useEffect Hooks of functional component are *componentDidMount(), componentDidUpdate(), and componentWillUnmount()*

* componentDidMount: is equivalent for running an *effect once*.  
  For example:

| useEffect(() => {  console.log("This is useEffect Hook equivalent of componentDidMount lifecycle method")  },[]); |
| --- |

**Note:** empty array = useEffect hook *runs once on mount*

* componentDidUpdate: is equivalent for *running effects when things change*For example:

| useEffect(() => {  console.log("The name props has changed!");  }, [props.name]); |
| --- |

* componentWillUnmount: To run a *hook as the component is about to unmount*, we just have to return a function from the useEffect Hook  
  For example:

| useEffect(() => {  console.log('running effect');  return () => {  console.log('unmount');  }  }) |
| --- |

*81*: **Compare useState and useReducer implementations** Related To: [React Hooks](https://www.fullstack.cafe/interview-questions/react-hooks)

* useState updates state with setState, while useReducer with *dispatch function*.
* useState passes down all the setState custom helper functions, while useReducer passes down just the *dispatch function*.
* useState needs to wrap functions in useCallback(if we want to memorize them), while *dispatch function* is already **memorized**.
* useState easier to write, useReducer is harder to implement and needs more logic to be coded.

*82*: **Do React Hooks cover all use cases for class components?**

No, The following methods have not been introduced in Hooks yet:

* getSnapshotBeforeUpdate
* getDerivedStateFromError
* componentDidCatch

*83*: **How can I make use of *Error Boundaries* in functional React components?**

As of v16.2.0, there's no way to turn a functional component into an error boundary. The componentDidCatch() method works like a JavaScript catch {} block, but for components. **Only class components can be error boundaries. In practice, most of the time you’ll want to declare an error boundary component once and use it throughout your application.**

Also bear in mind that try/catch blocks *won't work on all cases*. If a component deep in the hierarchy tries to update and fails, the try/catch block in one of the parents won't work -- because it isn't necessarily updating together with the child.

A few third party packages on npm implement error boundary hooks.

*84*: **When would you use useRef?** Related To: [React Hooks](https://www.fullstack.cafe/interview-questions/react-hooks)

The main use cases:

1 To store a ref to **DOM** elements so you can later do something with them:

| function TextInputWithFocusButton() {  const inputEl = useRef(null);  const onButtonClick = () => {  inputEl.current.focus();  };  return (  <>  <input ref={inputEl} type="text"/>  <button onClick={onButtonClick}>Focus the input</button>  </>  );  } |
| --- |

2 To store values *without* triggering *re-renders:*

| function Counter(){  const [count, setCount] = useState(0);  const prevCountRef = useRef();  useEffect(() => {  prevCountRef.current = count;  });  const prevCount = prevCountRef.current;  return <h1>Now: {count}, before: {prevCount} </h1>;  } |
| --- |

*85*: **How would you pass data from child to parent component in React?**

Often, several components need to reflect the same changing data. In React, sharing state is accomplished by **moving it up** to the closest common ancestor of the components that need it. This is called **lifting state up**.

A common technique for these situations is to **lift the state up** to the first common ancestor of all the components that need to use the state (i.e. the PageComponent in this case) and pass down the *state* and *state-altering functions* to the child components as props.

| function PageComponent() {  const [count, setCount] = useState(0);  const increment = () => {  setCount(count + 1)  }  return (  <div className="App">  <ChildComponent onClick={increment} count={count} />  <h2>count {count}</h2>  (count should be updated from child)  </div>  );  }  const ChildComponent = ({ onClick, count }) => {  return (  <button onClick={onClick}>  Click me {count}  </button>  )  }; |
| --- |

*86***: What is the purpose of super(props)?**

When you pass props to super, the props get assigned to this. Take a look at the following scenario:

| constructor(props) {  super();  console.log(this.props) //undefined  } |
| --- |

How ever when you do :

| constructor(props) {  super(props);  console.log(this.props) //props will get logged.  } |
| --- |

Note that passing or not passing props to super has no effect on later uses of this.props outside constructor. That is render, shouldComponentUpdate, or event handlers always have access to it.

*87*: **Explain the *Virtual DOM* concept in React**

There are in fact 2 problems that need to be solved here

1. **When do I re-render?** Answer: When I observe that the data is dirty.
2. **How do I re-render efficiently?** Answer: Using a virtual DOM to generate a real DOM patch

In React, each of your components have a state. This state is like an observable you might find in knockout or other MVVM style libraries. 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 you 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. You could re-render using a virtual DOM with or without dirty checking. With ReactJS, each time a change is triggered (by a query or a user’s action, for instance), the entire virtual DOM is updated. ReactJS keeps two versions of the virtual DOM in memory — an updated virtual DOM and a copy made before the update. After the update, ReactJS compares these two versions to find the elements that have changed. Then it updates only the part of the real DOM that has changed. There is some overhead in computing the diff between two virtual trees, but the virtual DOM diff is about understanding what needs updating in the DOM and not whether or not your data has changed.

*88*: **Describe *Flux* vs *MVC*?** Related To: [ASP.NET MVC](https://www.fullstack.cafe/interview-questions/aspnet-mvc)

Traditional MVC patterns have worked well for separating the concerns of data (Model), UI (View) and logic (Controller) — but MVC architectures frequently encounter two main problems:

* **Poorly defined data flow:** The cascading updates which occur across views often lead to a tangled web of events which is difficult to debug.
* **Lack of data integrity:** Model data can be mutated from anywhere, yielding unpredictable results across the UI.

With the Flux pattern complex UIs no longer suffer from cascading updates; any given React component will be able to reconstruct its state based on the data provided by the store. The Flux pattern also enforces data integrity by restricting direct access to the shared data.

*89*: **Can you force a React component to rerender without calling setState?**

In your component, you can call this.forceUpdate() to force a rerender. Another way is this.setState(this.state);

Note that forceUpdate should be avoided because it deviates from a React mindset. The React docs cite an example of when forceUpdate might be used:

By default, when your component's state or props change, your component will re-render. However, if these change implicitly (eg: data deep within an object changes without changing the object itself) or if your render() method depends on some other data, you can tell React that it needs to re-run render() by calling forceUpdate().

*90*: **Why does React use SyntheticEvents?**

React implements a **synthetic events** system that brings consistency and high performance to React apps and interfaces. It achieves consistency by normalizing events so that they have the same properties across different browsers and platforms.

A synthetic event is a cross-browser wrapper around the browser’s native event. It has the same interface as the browser’s native event, including stopPropagation() and preventDefault(), except the events work identically across all browsers.

It achieves high performance by automatically using event delegation. In actuality, React doesn’t attach event handlers to the nodes themselves. Instead, a single event listener is attached to the root of the document. When an event is fired, React maps it to the appropriate component element.

*91*: **How would you go about investigating slow React application rendering?**

One of the most common issues in React applications is when components *re-render unnecessarily*. There are two tools provided by React that are helpful in these situations:

* React.memo(): This prevents unnecessary re-rendering of function components
* PureComponent: This prevents unnecessary re-rendering of class components

Both of these tools rely on a shallow comparison of the props passed into the component—if the props have not changed, then the component will not re-render. While both tools are very useful, the shallow comparison brings with it an additional performance penalty, so both can have a negative performance impact if used incorrectly. By using the React Profiler, performance can be measured before and after using these tools to ensure that performance is actually improved by making a given change.

*92*: **What's a *Pure Functional Component* in React?**

A function is said to be pure if:

* Its return value is only determined by its input values
* Its return value is always the same for the same input values

A React component is considered pure if it renders the same output for the same state and props.

With React.memo(), you can create memoized functional components that bail out of rendering on unnecessary updates using shallow comparison of props.

*93***: 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 component has re-rendered.')  ) |
| --- |

*94***: When is it important to pass props to super(), and why?**

The only one reason when one needs to pass props to super() is when you want to access this.props in constructor:

| class MyComponent extends React.Component {  constructor(props) {  super(props)  console.log(this.props)  // -> { icon: 'home', … }  }  } |
| --- |

Not passing:

| class MyComponent extends React.Component {  constructor(props) {  super()  console.log(this.props)  // -> undefined  // Props parameter is still available  console.log(props)  // -> { icon: 'home', … }  }  render() {  // No difference outside constructor  console.log(this.props)  // -> { icon: 'home', … }  }  } |
| --- |

Note that passing or not passing props to super has no effect on later uses of this.props outside constructor.

*95*: **Why would you need to bind event handlers to this?**

Binding is not something that is specifc to React, but rather how this works in Javascript. When you define a component using an ES6 class, a common pattern is for an event handler to be a method on the class. In JavaScript, class methods are not bound by default. If you forget to bind this.someEventHandler and pass it to onChange, this will be undefined when the function is actually called.

Generally, if you refer to a method without () after it, such as onChange={this.someEventHandler}, you should bind that method.

*96***: What is the difference between using constructor vs getInitialState in React?**

The difference between constructor and getInitialState is the difference between ES6 and ES5 itself. You should initialize state in the constructor when using ES6 classes, and define the getInitialState method when using React.createClass.

| class MyComponent extends React.Component {  constructor(props) {  super(props);  this.state = { /\* initial state \*/ };  }  } |
| --- |

is equivalent to

| var MyComponent = React.createClass({  getInitialState() {  return { /\* initial state \*/ };  },  }); |
| --- |

*97*: **Why doesn't this.props.children.map work?**

this.props.children is an opaque data structure. It can be either an array or a single element. In your case, this.props.children is probably a single element, which is why the .map() method is undefined.

You should use the [React.Children API](https://reactjs.org/docs/react-api.html#reactchildren) when manipulating the children prop.

*98***: How to create *Props Proxy* for HOC component?**

You can add/edit props passed to the Component as a props proxy as below

| function HOC(WrappedComponent) {  return class Test extends Component {  render() {  const newProps = {  title: 'New Header',  footer: false,  showFeatureX: false,  showFeatureY: true  };  return <WrappedComponent {...this.props} {...newProps} />  }  }  } |
| --- |

*99***: How to *conditionally* add attributes to React components?**

**Problem**

**Is there a way to only add attributes to a React component if a certain condition is met?**

For certain attributes, React is intelligent enough to omit the attribute if the value you pass to it is not *truthy*. For example:

| var InputComponent = React.createClass({  render: function() {  var required = true;  var disabled = false;  return (  <input type="text" disabled={disabled} required={required} />  );  }  }); |
| --- |

will result in:

| <input type="text" required> |
| --- |

Another possible approach is:

| var condition = true;  var component = (  <div  value="foo"  { ...( condition && { disabled: true } ) } />  ); |
| --- |

*100*: **Does React re-render all components and sub components every time setState is called?**

By default - **yes**.

There is a method boolean shouldComponentUpdate(object nextProps, object nextState), each component has this method and it's responsible to determine "should component update (run render function)?" every time you change state or pass new props from parent component.

You can write your own implementation of shouldComponentUpdate method for your component, but default implementation always returns true - meaning always re-run render function.

*101*: **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.

*102*: **How to apply *validation* on props in ReactJS?**

When the application is running in development mode, React will automatically check for all props that we set on components to make sure they must right correct and right data type. For incorrect type, it will generate warning messages in the console for development mode whereas it is disabled in production mode due performance impact. The mandatory prop is defined with isRequired.

The set of predefined prop types are below

1. React.PropTypes.string
2. React.PropTypes.number
3. React.PropTypes.func
4. React.PropTypes.node
5. React.PropTypes.bool

For example, we define propTypes for user component as below,

| import PropTypes from 'prop-types';  class User extends React.Component {  render() {  return (  <h1>Welcome, {this.props.name}</h1>  <h2>Age, {this.props.age}  );  }  }  User.propTypes = {  name: PropTypes.string.isRequired,  age: PropTypes.number.isRequired  }; |
| --- |

*103*: **When would you use StrictMode component in React?**

I've found it especially useful to implement strict mode when I'm working on new code bases and I want to see what kind of code/components I'm facing. Also if you're on bug hunting mode, sometimes it's a good idea to wrap with <StrictMode /> the components/blocks of code you think might be the source of the problem.

*104***: What's the difference between useCallback and useMemo in practice? Related To:**

With useCallback you memoize functions, useMemo memoizes any computed value:

| const fn = () => 42 // assuming expensive calculation here  const memoFn = useCallback(fn, [dep]) // (1)  const memoFnReturn = useMemo(fn, [dep]) // (2) |
| --- |

(1) will return a memoized version of fn - same reference across multiple renders, as long as dep is the same. But *every time* **you** *invoke* **memoFn,** *that complex computation starts again***.**

(2) will invoke fn every time dep changes and remember its *returned value* (42 here), which is then stored in memoFnReturn.

*105*: **Explain why and when would you use useMemo()?**

**Why:**

In the lifecycle of a component, React re-renders the component when an update is made. When React checks for any changes in a component, it may detect an unintended or unexpected change due to how JavaScript handles equality and shallow comparisons. This change in the React application will cause it to re-render unnecessarily.

Additionally, if that re-rendering is an expensive operation, like a long for loop, it can hurt performance. Expensive operations can be costly in either time, memory, or processing.

**When:**

Optimal if the wrapped function is large and expensive.

**How:**

*Memoization* is an optimization technique which passes a complex function to be memoized. In memoization, the result is “remembered” when the same parameters are passed-in subsequently.

| const memoizedValue = React.useMemo(() => computeExpensiveValue(a, b), [a, b]); |
| --- |

useMemo takes in a function and an array of dependencies. The dependency’s list are the elements useMemo watches: if there are no changes, the function result will stay the same. Otherwise, it will re-run the function. If they don’t change, it doesn’t matter if our entire component re-renders, the function won’t re-run but instead return the stored result.

*106*: **When to use useCallback, useMemo and useEffect?** Related To: [React Hooks](https://www.fullstack.cafe/interview-questions/react-hooks)

* useEffect - It's the alternative for the class component lifecycle methods componentDidMount, componentWillUnmount, componentDidUpdate, etc. You can also use it to **create a side effect when dependencies change**, i.e. "If some variable changes, do this".
* useCallback - On every render, everything that's inside a functional component will run again. If a child component has a dependency on a function from the parent component, the child will re-render every time the parent re-renders even if that function "doesn't change" (the reference changes, but what the function does won't).  
  It's used for optimization by avoiding unnecessary renders from the child, making the function change the reference only when dependencies change. You should use it **when a function is a dependency of a side effect** e.g. useEffect.
* useMemo - It will run on every render, but with cached values. It will only use new values when certain dependencies change. It's used for optimization **when you have expensive computations**.

*107*: **Can you do *Components Inheritance* in React?**

The React Team has’t found any use cases where we would recommend creating component inheritance hierarchies.

Props and composition give you all the flexibility you need to customize a component’s look and behavior in an explicit and safe way. Remember that components may accept arbitrary props, including primitive values, React elements, or functions.

If you want to reuse non-UI functionality between components, we suggest extracting it into a separate JavaScript module. The components may import it and use that function, object, or a class, without extending it.

*108*: **What is difference between *Incremental DOM* and *Virtual DOM*?** Related To: [Angular](https://www.fullstack.cafe/interview-questions/angular)

* **Incremental DOM** is a library for building up DOM trees and updating them in-place when data changes. It differs from the established virtual DOM approach in that no intermediate tree is created (the existing tree is mutated in-place). This approach significantly reduces memory allocation and GC thrashing for incremental updates to the DOM tree therefore increasing performance significantly in some cases.
* **Virtual DOM** compares (diff) a new entire virtual DOM with the previous virtual DOM for changes then applies those changes to the actual DOM. - This approach creates a new virtual DOM to determine the changes (memory heavy). Has a big memory footprint because it needs headroom for changes that "might" happen to the virtual DOM.

*109*: **When would you use flushSync in ReactJS?**

React 18 adds out-of-the-box performance improvements by doing more **batching** (automated) by default. Batching is when React groups multiple state updates into a single re-render for better performance.

* To *opt-out* of automatic batching, you can use flushSync so your component will be re-rendered after each state update. You might need it when for example some code may depend on *reading* something from the *DOM immediately* after a state change.

Consider with automated batching:

| function handleClick() {  setCount(c => c + 1);  setFlag(f => !f);  // React will only re-render once at the end (that's batching!)  } |
| --- |

Consider with flushSync:

| import { flushSync } from 'react-dom';  function handleClick() {  flushSync(() => {  setCounter(c => c + 1);  });  // React has updated the DOM by now  flushSync(() => {  setFlag(f => !f);  });  // React has updated the DOM by now  } |
| --- |

*10***: When shall we use useReducer hook in ReactJS?**

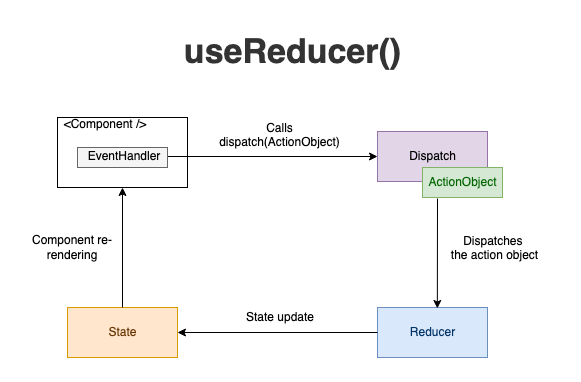
useReducer is an alternative to useState. useReducer is usually preferable to useState when you have complex state logic that involves multiple sub-values or when the next state *depends* on the previous one.

An example will be a list of items, where you need to add, update and remove items in the state, Here you might have noticed that the state management logic takes a good part of the component body. useReducer helps to separate the concerns of rendering vs a concern of state management.

useReducer also lets you optimize performance for components that trigger deep updates because [you can pass dispatch down instead of callbacks](https://reactjs.org/docs/hooks-faq.html#how-to-avoid-passing-callbacks-down).

| const initialState = {count: 0};  function reducer(state, action) {  switch (action.type) {  case 'increment':  return {count: state.count + 1};  case 'decrement':  return {count: state.count - 1};  default:  throw new Error();  }  }  function Counter() {  const [state, dispatch] = useReducer(reducer, initialState);  return (  <>  Count: {state.count}  <button onClick={() => dispatch({type: 'decrement'})}>-</button>  <button onClick={() => dispatch({type: 'increment'})}>+</button>  </>  );  } |
| --- |

React guarantees that dispatch function identity is stable and won’t change on re-renders. This is why it’s safe to omit from the useEffect or useCallback dependency list.



*111*: **When to use useState vs useReducer?** Related To: [React Hooks](https://www.fullstack.cafe/interview-questions/react-hooks)

The decision of whether to use useState or useReducer isn't always black and white; there are many shades of grey. But,

use useState if you have:

* JavaScript *primitives* as state
* *Simple* state transitions
* Business logic within your component
* Different properties that *don't change* in any correlated way and can be managed by multiple useState hooks

use useReducer if you have:

* JavaScript *objects* or *arrays* as state
* *Complex* state transitions
* *Complicated* business logic more suitable for a reducer function (to separate concern of it)
* Different properties tied together that should be managed in one state object (when state depends on state)

*112*: **How would you store *non-state*/*instance* variables in functional React components?**

You can use useRef hook (it's the recommended way stated in [docs](https://reactjs.org/docs/hooks-faq.html#is-there-something-like-instance-variables)). useRef returns an object whose reference would not change across re-renders, the actual value for foo is then kept in the current property of that object.

* Declaring variable: const a = useRef(5) // 5 is initial value
* getting the value: a.current
* setting the value: a.current = my\_value

*113*: **What is a *Pure Function*?**

A **Pure function** is a function that doesn't depend on and **doesn't modify the states** of variables **out of its scope**. Essentially, this means that a pure function will always return the same result given same parameters.

*114*: **Explain some difference between Flux and AngularJS (1.x) approach**

UI components in AngularJS typically rely on some internal $scope to store their data. This data can be directly mutated from within the UI component or anything given access to $scope — a risky situation for any part of the component or greater application which relies on that data.

By contrast, the Flux pattern encourages the use of immutable data. Because the store is the central authority on all data, any mutations to that data must occur within the store. The risk of data pollution is greatly reduced.

*115*: **What is the key architectural difference between a JavaScript library such as React and a JavaScript framework such as Angular?**

React enables developers to render a user interface. To create a full front-end application, developers need other pieces, such as state management tools like Redux.

Like React, Angular enables developers to render a user interface, but it is a “batteries included” framework that includes prescriptive, opinionated solutions to common requirements like state management.

While there are many other considerations when comparing React and Angular specifically, this key architectural difference means that:

* Using a library such as React can give a project a greater ability to evolve parts of the system—again for example, state management—over time, when new solutions are created by the open source community.
* Using a framework such as Angular can make it easier for developers to get started and can also simplify maintenance.

*116*: **What is React Fiber?**

React Fiber is an ongoing reimplementation of React's core algorithm. The main difference between react and react fiber are these new features :-

1. **Incremental Rendering :-** React v16.0 includes a completely rewritten server renderer. It’s really fast. It supports streaming, so you can start sending bytes to the client faster
2. **Handle errors in the render API :** To make class component an error boundary we define a new lifecycle method called componentDidCatch(error, info).
3. **Return multiple elements from render** : With this new feature in React v16.0 now we can also return an array of elements, and string from component’s render method.
4. **Portals** : Portals provide a first-class way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.
5. **Fragments** : A common pattern in React is for a component to return multiple elements. Fragments let you group a list of children without adding extra nodes to the DOM.

*117***: How to avoid the need for binding in React?**

There are several common approaches used to avoid methods binding in React:

1 Define Your Event Handler as an Inline Arrow Function

| class SubmitButton extends React.Component {  constructor(props) {  super(props);  this.state = {  isFormSubmitted: false  };  }  render() {  return (  <button onClick={() => {  this.setState({ isFormSubmitted: true });  }}>Submit</button>  )  }  } |
| --- |

Define Your Event Handler as an Arrow Function Assigned to a Class Field

| class SubmitButton extends React.Component {  state = {  isFormSubmitted: false  }  handleSubmit = () => {  this.setState({  isFormSubmitted: true  });  }  render() {  return (  <button onClick={this.handleSubmit}>Submit</button>  )  }  } |
| --- |

3 Use a Function Component with Hooks

| const SubmitButton = () => {  const [isFormSubmitted, setIsFormSubmitted] = useState(false);  return (  <button onClick={() => {  setIsFormSubmitted(true);  }}>Submit</button>  )  }; |
| --- |

*118*: **How does React renderer work exactly when we call setState?**

There are two steps of what we may call render:

1. Virtual DOM render: when render method is called it returns a new *virtual dom* structure of the component. This render method is called always when you call setState(), because shouldComponentUpdate always returns true by default. So, by default, there is no optimisation here in React.
2. Native DOM render: React changes real DOM nodes in your browser only if they were changed in the Virtual DOM and as little as needed - this is that great React's feature which optimizes real DOM mutation and makes React fast.

*119***: How to use React.memo()?**

With React.memo(), you can create memoized functional components that bail out of rendering on unnecessary updates using shallow comparison of props.

Using the new React.memo() API, the functional component can be wrapped as follows:

| import React, { memo } from 'react';  function PercentageStat({ label, score = 0, total = Math.max(1, score) }) {  return (  <div>  <h6>{ label }</h6>  <span>{ Math.round(score / total \* 100) }%</span>  </div>  )  }  // Wrap component using `React.memo()`  export default memo(PercentageStat); |
| --- |

*120*: **Can a custom React hook return JSX?** Related To: [React Hooks](https://www.fullstack.cafe/interview-questions/react-hooks)

While there is no hardcore restriction on how you should define custom hooks and what logic should contain, it's an anti-pattern to write hooks that return JSX.

There are a few downsides to using hooks to return JSX

* When you write a hook that returns JSX component, you are essentially defining the component within the functional component, so on each and every re-render you will be creating a new instance of the component. This will lead to the component being unmounted and mounted again. This is bad for performance and also buggy if you have stateful login within the component as the state will get reset with every re-render of the parent
* By defining a JSX component within the hook, you are taking away the option of lazy loading your component if the need be.
* Any performance optimization to the component will require you to make use of useMemo which doesn't give you the flexibility of a custom comparator function like React.memo()

The benefit on the other hand is that you have control over the state of the component in the parent. However, you can still implement the same logic by using a controlled component approach

*121*: **What is the order of useInsertionEffect, useEffect and useLayoutEffect hooks at component generation ?**

useInsertionEffect

It fires synchronously **before** all DOM mutations. Use this to inject styles into the DOM **before** reading layout in useLayoutEffect. So it runs before useLayoutEffect.

useLayoutEffect

It fires synchronously **after** all DOM mutations. Use this to read layout from the DOM and synchronously re-render.

useEffect

It will run after the render is committed to the screen. So it runs after useLayoutEffect.

Therefore the order of running is:

1. useInsertionEffect
2. useLayoutEffect
3. useEffect

**Coding Challenges**

*1***:** Provide an example of any simple *Custom* React Hook. Why do we need Custom Hooks?

A Custom Hook is a stateful function that uses other react built-in hooks (e.g. useState, useCallback etc.) that can wrap around the *stateful* logic that you wanted to gather in one place and *avoid copying and pasting the same logic* in multiple components.

Consider the increment/decriment custom hook:

| const useCounter = () => {  const [counter, setCounter] = useState(0);  return {  counter, // counter value  increment: () => setCounter(counter + 1), // function 1  decrement: () => setCounter(counter - 1) // function 2  };  }; |
| --- |

**and then in your component you can use it as follows:**

| const Component = () => {  const { counter, increment, decrement } = useCounter();  return (  <div>  <span onClick={decrement}>-</span>  <span style={{ padding: "10px" }}>{counter}</span>  <span onClick={increment}>+</span>  </div>  );  } |
| --- |

*2***: What is *equivalent* of the following using React.createElement?**

**Question:**

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

**What is equivalent of the following using React.createElement?**

**Answer:**

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

*3*: **Given the React code defined above, can you identify two problems?**

**Problem**

Take a look at the code below:

| class MyComponent extends React.Component {  constructor(props) {  // set the default internal state  this.state = {  clicks: 0  };  }  componentDidMount() {  this.refs.myComponentDiv.addEventListener('click', this.clickHandler);  }  componentWillUnmount() {  this.refs.myComponentDiv.removeEventListener('click', this.clickHandler);  }  clickHandler() {  this.setState({  clicks: this.clicks + 1  });  }  render() {  let children = this.props.children;  return (  <div className="my-component" ref="myComponentDiv">  <h2>My Component ({this.state.clicks} clicks})</h2>  <h3>{this.props.headerText}</h3>  {children}  </div>  );  }  } |
| --- |

Given the code defined above, can you identify two problems?

**Answer**

1 The constructor does not pass its props to the super class. It should include the following line:

| constructor(props) {  super(props);  // ...  } |
| --- |

The event listener (when assigned via addEventListener()) is not properly scoped because [ES2015 doesn’t provide autobinding](https://facebook.github.io/react/docs/reusable-components.html#no-autobinding). Therefore the developer can re-assign clickHandler in the constructor to include the correct binding to this:

| constructor(props) {  super(props);  this.clickHandler = this.clickHandler.bind(this);  // ...  } |
| --- |

*4***: What is *equivalent* of this code using React Hooks?**

**Problem**

Let's say in our project we have componentWillUnmount that is used for cleanup (like removing event listeners, cancel the timer etc). How to refactor this code using React Hooks?

| componentDidMount() {  window.addEventListener('mousemove', () => {})  }  componentWillUnmount() {  window.removeEventListener('mousemove', () => {})  } |
| --- |

**Answer**

React Hooks equivalent of above code will be as follows

| useEffect(() => {  window.addEventListener('mousemove', () => {});  // returned function will be called on component unmount  return () => {  window.removeEventListener('mousemove', () => {})  }  }, []) |
| --- |

*5*: **What's *wrong* with that code?**

**Problem**

Consider:

| this.setState({  counter: this.state.counter + this.props.increment,  }); |
| --- |

What's wrong with that code?

**Answer**

Because this.props and this.state may be updated asynchronously, you should not rely on their values for calculating the next state. To fix it, use a second form of setState() that accepts a function rather than an object. That function will receive the previous state as the first argument, and the props at the time the update is applied as the second argument:

| // Correct  this.setState((state, props) => ({  counter: state.counter + props.increment  })); |
| --- |

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

**Question:**

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'  // fetchUser take in a username returns a promise  // which will resolve with that username's data.  class Twitter extends Component {  // finish this  } |
| --- |

**Answer:**

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.

| 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)  }  } |
| --- |

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.

*7*: **What is wrong with this code?**

**Problem**

What is wrong with this code?

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

**Answer**

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.

*8*: **How to mitigate multiple component re-renders when using multiple useState calls?**

**Problem**

| Consider this code:  const getData = url => {  const [data, setData] = useState(null);  const [loading, setLoading] = useState(true);  useEffect(async () => {  const test = await api.get('/people')  if(test.ok){  setLoading(false);  setData(test.data.results);  }  }, []);  return { data, loading };  }; |
| --- |

When I get data and update two different state variables (data and loading flag), my component (a data table) is rendered twice, even though both calls to the state updater are happening in the same function. Any ideas on how to mitigate this?

**Answer**

**1.** You could combine the loading state and data state into one state object and then you could do one setState call and there will only be one render.

| const [userRequest, setUserRequest] = useState({  loading: false,  user: null,  });  useEffect(() => {  // Note that this replaces the entire object and deletes user key!  setUserRequest({ loading: true });  fetch('https://randomuser.me/api/')  .then(results => results.json())  .then(data => {  setUserRequest({  loading: false,  user: data.results[0],  });  });  }, []); |
| --- |

2. Before React 18, React will batch state updates if they're triggered from within a React-based event, like a button click or input change. It will not batch updates if they're triggered outside of a React event handler, like an async call.

3. Starting in React 18 (opt-in feature) all updates will be automatically batched, no matter where they originate from.

**9: How would you *optimise* this code using one of the React Hooks?**

**Problem**

Consider this code. Imagine there are 100000 of users. Do you see any optimization we can make?

| import React from 'react';    const users = [  { id: 'a', name: 'Robin' },  { id: 'b', name: 'Dennis' },  ...  ];    const App = () => {  const [text, setText] = React.useState('');  const [search, setSearch] = React.useState('');    const handleText = (event) => {  setText(event.target.value);  };    const handleSearch = () => {  setSearch(text);  };    const filteredUsers = users.filter((user) => {  return user.name.toLowerCase().includes(search.toLowerCase());  });    return (  <div>  <input type="text" value={text} onChange={handleText} />  <button type="button" onClick={handleSearch}>  Search  </button>    <List list={filteredUsers} />  </div>  );  };    const List = ({ list }) => {  return (  <ul>  {list.map((item) => (  <ListItem key={item.id} item={item} />  ))}  </ul>  );  };    const ListItem = ({ item }) => {  return <li>{item.name}</li>;  };    export default App; |
| --- |

Tip: try to find an use case for useMemo hook

**Answer**

Even though the filteredUsers don't change when someone types into the input field, because they change only when clicking the button via the search state, the filter's callback function runs again and again for every keystroke in the input field.

| function App() {  ...    const filteredUsers = users.filter((user) => {  console.log('Filter function is running ...');  return user.name.toLowerCase().includes(search.toLowerCase());  });    ...  } |
| --- |

This doesn't slow down this small React application. However, if we would deal with a large set of data in this array and run the filter's callback function for every keystroke, we would maybe slow down the application.

you can use React's useMemo Hook to memoize a functions return value(s) and to run a function only if its dependencies (here search) have changed:

| function App() {  ...    const filteredUsers = React.useMemo(  () =>  users.filter((user) => {  console.log('Filter function is running ...');  return user.name.toLowerCase().includes(search.toLowerCase());  }),  [search]  );    ...  } |
| --- |

**React Hooks**

*Q1*: **What are React Hooks?** Related To: [React](https://www.fullstack.cafe/interview-questions/react)

**Hooks** are a new addition in 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.

*2***: What is useState() in React? Related To:** [**React**](https://www.fullstack.cafe/interview-questions/react)

**Problem**

Explain what is the use of useState(0) there:

| ...  const [count, setCounter] = useState(0);  const [moreStuff, setMoreStuff] = useState(...);  ...  const setCount = () => {  setCounter(count + 1);  setMoreStuff(...);  ...  }; |
| --- |

**Answer**

useState is one of build-in react hooks. useState(0) returns a tuple where the first parameter count is the current state of the counter and setCounter is the method that will allow us to update the counter's state.

We can use the setCounter method to update the state of count anywhere - In this case we are using it inside of the setCount function where we can do more things; the idea with hooks is that we are able to keep our code more functional and avoid class based components if not desired/needed.

*3*: **What are advantages of using React Hooks?** Related To: [React](https://www.fullstack.cafe/interview-questions/react)

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 to easily manipulate the state of our functional component 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 .

*4***: How to call loading function with React useEffect only once? Related To:** [**React**](https://www.fullstack.cafe/interview-questions/react)

If you only want to run the function given to useEffect after the *initial render*, you can give it an *empty array* [] as the second argument.

For example:

| function MyComponent(){  useEffect(() => {  loadDataOnlyOnce();  }, []);  return <div> { /\*...\*/} </div>;  } |
| --- |

*5*: **How to access DOM elements in React?** Related To: [React](https://www.fullstack.cafe/interview-questions/react)

One of the useful application of the useRef() hook is to access DOM elements. This is performed in 3 steps:

1. Define the reference to access the element const elementRef = useRef();
2. Assign the reference to ref attribute of the element: <div ref={elementRef}></div>;
3. After mounting, elementRef.current points to the DOM element.

Consider:

| import { useRef, useEffect } from 'react';  function AccessingElement() {  const elementRef = useRef();  useEffect(() => {  const divElement = elementRef.current;  console.log(divElement); // logs <div>I'm an element</div>  }, []);  return (  <div ref={elementRef}>  I'm an element  </div>  );  } |
| --- |

*6*: **How to use componentWillMount() in React Hooks?**

You cannot use any of the existing lifecycle methods (componentDidMount, componentDidUpdate, componentWillUnmount etc.) in a hook. They can only be used in class components. And with Hooks you can only use in functional components.

You can think of useEffect Hook as componentDidMount, componentDidUpdate, and componentWillUnmount combined.

1 Code inside **componentDidMount** run once when the component is mounted. useEffect hook equivalent for this behaviour is

| useEffect(() => {  // Your code here  }, []); |
| --- |

2 Without the second parameter the useEffect hook will be called on every render (like componentDidUpdate) of the component which can be dangerous:

| useEffect(() => {  // Your code here  }); |
| --- |

3 Hook equivalent of componentWillUnmount() code will be as follows

| useEffect(() => {  window.addEventListener('mousemove', () => {});  // returned function will be called on component unmount  return () => {  window.removeEventListener('mousemove', () => {})  }  }, []) |
| --- |

*7*: **Does React useState Hook update *immediately*?**

**Problem**

And how do you perform an action after useState hook has triggered?

**Answer**

React useState and setState don’t make changes directly to the state object; they create queues to optimize performance, which is why the changes don’t update immediately. The process to update React state is asynchronous for performance reasons.

To perform side effects after state has change, you must use the useEffect

*8***: What does *Batching* mean in ReactJS?**

Batching is nothing but grouping React multiple state updates together into a single render state to achieve better computational performance. Until React 18, we only *batched* updates during the *React event handlers*. Updates inside of promises, setTimeout, *native event handlers*, or any other event were not *batched* in React by default.

| function App() {  const [count, setCount] = useState(0);  const [flag, setFlag] = useState(false);  function handleClick() {  setCount(c => c+1); // Does not re-render yet  setFlag(f => !f); // Does not re-render yet  // React will only re-render once at the end (that's batching!)  }  return (  <div>  <button onClick={handleClick}>Next</button>  <h1 style={{color: flag ? "blue": "black"}}>{count}</h1>  </div>  )  } |
| --- |

*9*: **What are the advantages of *Batching* in ReactJS?** Related To: [React](https://www.fullstack.cafe/interview-questions/react)

* **Batching** is great for *performance* because it avoids unnecessary re-renders.
* **Batching** also *prevents* your component from *rendering* half-finished states where only one state variable was updated, which may cause bugs.
* Another reason to use **batching** is when the web application grows, the number of nested components will increase. Therefore, if a parent component executes an *unbatched state updated*, the *entire component tree will be re-rendered per state update* that is expensive.

*10*: **Which *lifecycle* methods of class component is replaced by useEffect in functional component?** Related To: [React](https://www.fullstack.cafe/interview-questions/react)

The lifecyce methods replaced by useEffect Hooks of functional component are *componentDidMount(), componentDidUpdate(), and componentWillUnmount()*

1 componentDidMount: is equivalent for running an *effect once*.  
For example:

| useEffect(() => {  console.log("This is useEffect Hook equivalent of componentDidMount lifecycle method")  },[]); |
| --- |

**Note:** empty array = useEffect hook *runs once on mount*

2 componentDidUpdate: is equivalent for *running effects when things change*For example:

| useEffect(() => {  console.log("The name props has changed!");  }, [props.name]); |
| --- |

3 componentWillUnmount: To run a *hook as the component is about to unmount*, we just have to return a function from the useEffect Hook  
For example:

| useEffect(() => {  console.log('running effect');  return () => {  console.log('unmount');  }  }) |
| --- |

*11***: Compare React Context Api with useContext React hook**

**When using the React Context API:**

* We need to wrap our content in a *Consumer component* and then *pass a function* as a child just so we could *access (or consume) our state*.
* This introduces *unnecessary component nesting* and *increases the complexity* of our code.

**Consider:**

| import React from "react";  import ReactDOM from "react-dom";  // Create a Context  const NumberContext = React.createContext();  // It returns an object with 2 values:  // { Provider, Consumer }  function App() {  // Use the Provider to make a value available to all  // children and grandchildren  return (  <NumberContext.Provider value={42}>  <div>  <Display />  </div>  </NumberContext.Provider>  );  }  function Display() {  // Use the Consumer to grab the value from context  // Notice this component didn't get any props!  return (  <NumberContext.Consumer>  {value => <div>The answer is {value}.</div>}  </NumberContext.Consumer>  );  }  ReactDOM.render(<App />, document.querySelector("#root")); |
| --- |

**When using useContext Hook:**

* lets you “use” context without a Consumer:

**Consider:**

| const newContext = React.createContext({ color: 'black' });  const value = useContext(newContext);  console.log(value); // this will return { color: 'black' } |
| --- |

*12***: When would you use useContext hook?**

React’s useContext hook makes it easy to pass data throughout your app without manually passing props down the tree. React Context is a way to manage state globally.

Consider:

| import { useState, createContext } from "react";  import ReactDOM from "react-dom/client";  const UserContext = createContext() |
| --- |

Wrap child components in the Context Provider and supply the state value.

| function Component1() {  const [user, setUser] = useState("Jesse Hall");  return (  <UserContext.Provider value={user}>  <h1>{`Hello ${user}!`}</h1>  <Component2 user={user} />  </UserContext.Provider>  );  } |
| --- |

Then you can access the user Context in all components:

| import { useState, createContext, useContext } from "react";  function Component5() {  const user = useContext(UserContext);  return (  <>  <h1>Component 5</h1>  <h2>{`Hello ${user} again!`}</h2>  </>  );  } |
| --- |

*13*: **Is there any problem when using useContext Hook?**

* The problem is that any *component consuming* state with useContext will **re-render** when any piece of the Context’s state updates. This resulted in *components that were totally divorced* from one another causing each other to *re-render*.
* In cases where these re-renders were expensive, the memory in *users’ browsers accumulated JS Heap footprints* in the orders of gigabytes.

*14*: **Compare useState and useReducer implementations** Related To: [React](https://www.fullstack.cafe/interview-questions/react)

* useState updates state with setState, while useReducer with *dispatch function*.
* useState passes down all the setState custom helper functions, while useReducer passes down just the *dispatch function*.
* useState needs to wrap functions in useCallback(if we want to memorize them), while *dispatch function* is already **memorized**.
* useState easier to write, useReducer is harder to implement and needs more logic to be coded.

*15*: **Do React Hooks cover all use cases for class components?** Related To: [React](https://www.fullstack.cafe/interview-questions/react)

No, The following methods have not been introduced in Hooks yet:

* getSnapshotBeforeUpdate
* getDerivedStateFromError
* componentDidCatch

*16*: **How can I make use of *Error Boundaries* in functional React components?**

As of v16.2.0, there's no way to turn a functional component into an error boundary. The componentDidCatch() method works like a JavaScript catch {} block, but for components. **Only class components can be error boundaries. In practice, most of the time you’ll want to declare an error boundary component once and use it throughout your application.**

Also bear in mind that try/catch blocks *won't work on all cases*. If a component deep in the hierarchy tries to update and fails, the try/catch block in one of the parents won't work -- because it isn't necessarily updating together with the child.

A few third party packages on npm implement error boundary hooks.

*17*: **What are differences between React.memo() and useMemo()?**

* React.memo() is a higher-order component (HOC) that we can use to wrap **components** that we do not want to re-render unless props within them change
* useMemo() is a React Hook that we can use to wrap **functions** within a component. We can use this to ensure that the values within that function are re-computed only when one of its dependencies change

*18***: What are common use cases for the useMemo?**

**The primary purpose of useMemo hook is** *"performance optimization"***.**

* It returns a *memoized value*,
* It accepts *two arguments* - ***create*** *function* (which should return a value to be memoized) and ***dependency*** *array*. It will *recompute the memoized value only when one of the dependencies has changed.*

**Using useMemo you achieve:**

* referential equality of the values (to further send them to props of the components to potentially avoid re-renders)
* eliminate redoing of the computationally expensive operations for same parameters

**For example:**

| function App() {  const [data, setData] = useState([....]);  function format() {  console.log('formatting....'); // this will print only when data has changed  const formattedData = [];  data.forEach(item => {  const newItem = //...do soemthing here,  if (newItem) {  formattedData.push(newItem);  }  })  return formattedData;  }  const formattedData = useMemo(format, [data])  return (  <>  {formattedData.map(item => (  <div key={item.id}>  {item.title}  </div>  ))}  </>  )  } |
| --- |

*19*: **What are production use cases for the useRef?**

* useRef simply returns a *plain Javascript object*. Its value can be *accessed* and *modified* (mutability) as many times as you need without worrying about *rerender*.
* useRef value will *persist* (won't be reset to the *initialValue* unlike an ordinary object defined in your function component; it persists because useRef gives you the same object *instead of creating a new one* on subsequent renders) for the component lifetime and across re-renders.
* useRef hook is often used to store values instead of DOM references. These values can either be a state that does not need to change too often or a state that should change as frequently as possible but should *not trigger* full re-rendering of the component.

For example:

| const refObject = useRef(initialValue); |
| --- |

*20*: **Explain the difference between useState and useRef hooks?**

1. Updating a reference created by useRef doesn't trigger re-rendering, while updating the state (setState) makes the component re-render;
2. useRef returns an *object with a current property holding the actual value*. In contrast, useState returns an *array with two elements*.
3. useRef‘s current property is *mutable*, but useState‘s state variable is *not*.
4. The reference update is *synchronous* (the updated reference value is available right away), while the state update is *asynchronous* (the state variable is updated after re-rendering).

Using useRef - no re-renders

| const countRef = useRef(0);    const handle = () => {  countRef.current++;  console.log(`Clicked ${countRef.current} times`);  }; |
| --- |

Using useState - triggers re-render

| const [count, setCount] = useState(0);    const handle = () => {  const updatedCount = count + 1;  console.log(`Clicked ${updatedCount} times`);  setCount(updatedCount);  }; |
| --- |

*1***: When would you use useRef?**

**The main use cases:**

1 To store a ref to **DOM** elements so you can later do something with them:  
Consider:

| function TextInputWithFocusButton() {  const inputEl = useRef(null);  const onButtonClick = () => {  inputEl.current.focus();  };  return (  <>  <input ref={inputEl} type="text"/>  <button onClick={onButtonClick}>Focus the input</button>  </>  );  } |
| --- |

2 To store values *without* triggering *re-renders:*Consider:

| function Counter(){  const [count, setCount] = useState(0);  const prevCountRef = useRef();  useEffect(() => {  prevCountRef.current = count;  });  const prevCount = prevCountRef.current;  return <h1>Now: {count}, before: {prevCount} </h1>;  } |
| --- |

*22*: **When writing a Custom Hook, what is the difference between it and a normal function?**

Hooks use a *stateful* closure around the invocation of the function component to store the state on behalf of that component instance. That closure is maintained by React.

* Custom hook will only be "stateful" if you use state with useState inside (or something that implements useState),
* Hooks should be called from the React code only not from the regular JS functions. Hence, Hooks' scope is limited to the React code world and has more power to do a lot with React code,
* In the class-based components, the Hooks won't work but regular functions will.
* In the regular JS functions, you can't access useState, useEffect, useContext etc. but in react custom hooks I can.

*23*: **Do two components using the same Hook share state?**

No. Custom Hooks are a mechanism to reuse *stateful logic* (such as setting up a subscription and remembering the current value), but every time you use a custom Hook, all state and effects inside of it are *fully isolated*.

*24*: **Do Hooks replace render props and higher-order components (HOC)?**

Often, render props and higher-order components render only a single child. React team thinks *Hooks are a simpler way to serve this use case*.

**There is still a place for both patterns** (for example, a virtual scroller component might have a renderItem prop, or a visual container component might have its own DOM structure). But in most cases, Hooks will be sufficient and can help reduce nesting in your tree.

*25***: What's the difference between useCallback and useMemo in practice?**

With useCallback you memoize functions, useMemo memoizes any computed value:

| const fn = () => 42 // assuming expensive calculation here  const memoFn = useCallback(fn, [dep]) // (1)  const memoFnReturn = useMemo(fn, [dep]) // (2) |
| --- |

(1) will return a memoized version of fn - same reference across multiple renders, as long as dep is the same. But *every time* **you** *invoke* **memoFn,** *that complex computation starts again***.**

(2) will invoke fn every time dep changes and remember its *returned value* (42 here), which is then stored in memoFnReturn.

*26*: **Explain why and when would you use useMemo()?** Related To: [React](https://www.fullstack.cafe/interview-questions/react)

**Why:**

In the lifecycle of a component, React re-renders the component when an update is made. When React checks for any changes in a component, it may detect an unintended or unexpected change due to how JavaScript handles equality and shallow comparisons. This change in the React application will cause it to re-render unnecessarily.

Additionally, if that re-rendering is an expensive operation, like a long for loop, it can hurt performance. Expensive operations can be costly in either time, memory, or processing.

**When:**

Optimal if the wrapped function is large and expensive.

**How:**

*Memoization* is an optimization technique which passes a complex function to be memoized. In memoization, the result is “remembered” when the same parameters are passed-in subsequently.

| const memoizedValue = React.useMemo(() => computeExpensiveValue(a, b), [a, b]); |
| --- |

useMemo takes in a function and an array of dependencies. The dependency’s list are the elements useMemo watches: if there are no changes, the function result will stay the same. Otherwise, it will re-run the function. If they don’t change, it doesn’t matter if our entire component re-renders, the function won’t re-run but instead return the stored result.

*27***: How do I update state on a *nested object* with useState()?**

**Problem**

I have a component that receives a prop that looks like this:

| const styles = {  font: {  size: {  value: '22',  unit: 'px'  },  weight: 'bold',  color: '#663300',  family: 'arial',  align: 'center'  }  }; |
| --- |

**How to update only the align property?**

**Answer**

You need to use *spread syntax*. Also while trying to update current state based on previous, use the callback pattern os setState:

| const { ...styling } = styles;  const [style, setStyle] = useState(styling);  ...  setStyle(prevStyle => ({  ...prevStyle,  font: { ...prevStyle.font, align: event.target.value }  })); |
| --- |

*Q28*: **When to use useCallback, useMemo and useEffect?**

* useEffect - It's the alternative for the class component lifecycle methods componentDidMount, componentWillUnmount, componentDidUpdate, etc. You can also use it to **create a side effect when dependencies change**, i.e. "If some variable changes, do this".
* useCallback - On every render, everything that's inside a functional component will run again. If a child component has a dependency on a function from the parent component, the child will re-render every time the parent re-renders even if that function "doesn't change" (the reference changes, but what the function does won't).  
  It's used for optimization by avoiding unnecessary renders from the child, making the function change the reference only when dependencies change. You should use it **when a function is a dependency of a side effect** e.g. useEffect.
* useMemo - It will run on every render, but with cached values. It will only use new values when certain dependencies change. It's used for optimization **when you have expensive computations**.

*29*: **When would you use flushSync in ReactJS?**

React 18 adds out-of-the-box performance improvements by doing more **batching** (automated) by default. Batching is when React groups multiple state updates into a single re-render for better performance.

* To *opt-out* of automatic batching, you can use flushSync so your component will be re-rendered after each state update. You might need it when for example some code may depend on *reading* something from the *DOM immediately* after a state change.

Consider with automated batching:

| function handleClick() {  setCount(c => c + 1);  setFlag(f => !f);  // React will only re-render once at the end (that's batching!)  } |
| --- |

Consider with flushSync:

| import { flushSync } from 'react-dom';  function handleClick() {  flushSync(() => {  setCounter(c => c + 1);  });  // React has updated the DOM by now  flushSync(() => {  setFlag(f => !f);  });  // React has updated the DOM by now  } |
| --- |

*30***: When shall we use useReducer hook in ReactJS? Related To:** [**React**](https://www.fullstack.cafe/interview-questions/react)

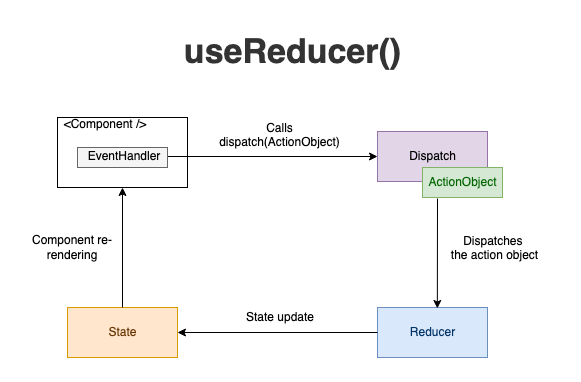
useReducer is an alternative to useState. useReducer is usually preferable to useState when you have complex state logic that involves multiple sub-values or when the next state *depends* on the previous one.

An example will be a list of items, where you need to add, update and remove items in the state, Here you might have noticed that the state management logic takes a good part of the component body. useReducer helps to separate the concerns of rendering vs a concern of state management.

useReducer also lets you optimize performance for components that trigger deep updates because [you can pass dispatch down instead of callbacks](https://reactjs.org/docs/hooks-faq.html#how-to-avoid-passing-callbacks-down).

| const initialState = {count: 0};  function reducer(state, action) {  switch (action.type) {  case 'increment':  return {count: state.count + 1};  case 'decrement':  return {count: state.count - 1};  default:  throw new Error();  }  }  function Counter() {  const [state, dispatch] = useReducer(reducer, initialState);  return (  <>  Count: {state.count}  <button onClick={() => dispatch({type: 'decrement'})}>-</button>  <button onClick={() => dispatch({type: 'increment'})}>+</button>  </>  );  } |
| --- |

React guarantees that dispatch function identity is stable and won’t change on re-renders. This is why it’s safe to omit from the useEffect or useCallback dependency list



*31*: **When to use useState vs useReducer?** Related To: [React](https://www.fullstack.cafe/interview-questions/react)

The decision of whether to use useState or useReducer isn't always black and white; there are many shades of grey. But,

use useState if you have:

* JavaScript *primitives* as state
* *Simple* state transitions
* Business logic within your component
* Different properties that *don't change* in any correlated way and can be managed by multiple useState hooks

use useReducer if you have:

* JavaScript *objects* or *arrays* as state
* *Complex* state transitions
* *Complicated* business logic more suitable for a reducer function (to separate concern of it)
* Different properties tied together that should be managed in one state object (when state depends on state)

*32*: **Is there a React hook equivalent to componentDidCatch?**

**Problem**

And how can I add **Error Boundary** to the functional component?

**Answer**

There is not a React hook equivalent of componentDidCatch. However, the React team plans to add one soon.

For functional components you can sometimes use try...catch to catch component errors. Also bear in mind that try/catch blocks *won't work on all cases*. If a component deep in the hierarchy tries to updates and fails, the try/catch block in one of the parents won't work -- because it isn't necessarily updating together with the child.

| function myComponent(...) {  // Initialize state variables.  // Initialize context variables.  // Initialize custom hooks.  // ...  try {  // Define internal functions.  // Define internal variables.  return (  <SomeThing />  )  } catch (error) {  // Catch internal functions, variables and return (jsx) errors.  // You can also create a lib to log the error to an error reporting service  // and use it here.  }  } |
| --- |

The try...catch block is the **Error Boundry** here.

**Note**: that approach has some limitations: when you use a hook like useEffect and you use some internal functions in it, you cannot put that internal function into try...catch(Error Boundary) block because you should define that function on top of useEffect hook ([why?](https://stackoverflow.com/questions/56850196/where-should-i-declare-functions-that-are-called-inside-a-useeffect-hook)) and you shouldn't use useEffect conditionally ([why?](https://reactjs.org/docs/hooks-rules.html#only-call-hooks-at-the-top-level))!

*33***: Are there any *problems* using useCallback?**

**Answer**

Look at the common anti-pattern of using useCallback:

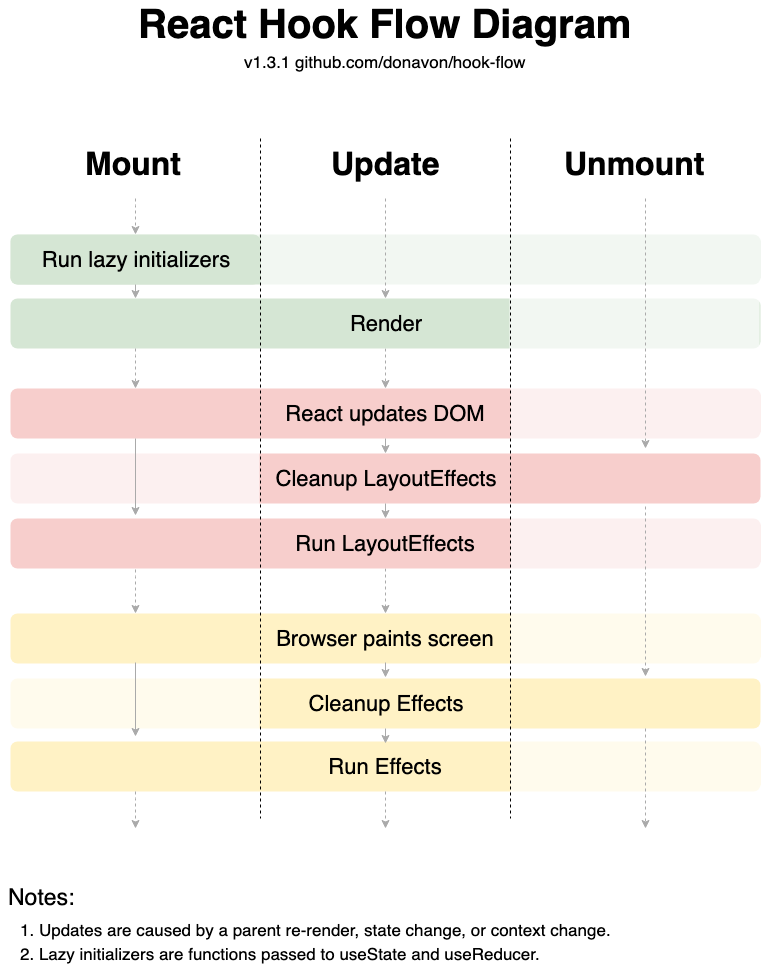
| import { useCallback } from 'react';  function MyComponent() {  // Contrived use of `useCallback()`  const handleClick = useCallback(() => {  // handle the click event  }, []);  return <MyChild onClick={handleClick}/>;  }  function MyChild({ onClick }) {  return <button onClick={onClick}>I am a child</button>;  } |
| --- |

Since *MyChild* component is *light* and its *re-rendering doesn't create performance issues*, using useCallback here is not important. Using useCallback you also increased code *complexity*.

*34*: **Explain the use of useLayoutEffect React Hook**

useLayoutEffect React Hook runs synchronously immediately after React has performed all DOM mutations. This can be useful if you need to make DOM measurements (like getting the scroll position or other styles for an element) and then make DOM mutations **or** trigger a synchronous re-render by updating the state.

As far as scheduling and lifecycle, this works the same way as componentDidMount and componentDidUpdate. Your code runs immediately after the DOM has been updated, but before the browser has had a chance to "paint" those changes (the user doesn't actually see the updates until after the browser has repainted).



*35*: **How would you store *non-state*/*instance* variables in functional React components?** Related To: [React](https://www.fullstack.cafe/interview-questions/react)

You can use useRef hook (it's the recommended way stated in [docs](https://reactjs.org/docs/hooks-faq.html#is-there-something-like-instance-variables)). useRef returns an object whose reference would not change across re-renders, the actual value for foo is then kept in the current property of that object.

* Declaring variable: const a = useRef(5) // 5 is initial value
* getting the value: a.current
* setting the value: a.current = my\_value

*Q36***: How can I *force* component to re-render with Hooks in React?**

This is possible with useState or useReducer, since [useState uses useReducer internally](https://github.com/facebook/react/blob/16.8.6/packages/react-dom/src/server/ReactPartialRendererHooks.js#L254):

| const [, updateState] = React.useState();  const forceUpdate = React.useCallback(() => updateState({}), []); |
| --- |

useCallback memoizes forceUpdate, so it stays constant during component lifespan and can be passed as a prop safely. updateState({}) updates the state with new object on each forceUpdate call, this results in a re-render. So yes, it forces an update when being called.

**Note**: forceUpdate isn't intended to be used under normal circumstances, only in testing or other outstanding cases. This situation may be addressed in a more conventional way.

*37*: **Can a custom React hook return JSX?** Related To: [React](https://www.fullstack.cafe/interview-questions/react)

While there is no hardcore restriction on how you should define custom hooks and what logic should contain, it's an anti-pattern to write hooks that return JSX.

There are a few downsides to using hooks to return JSX

* When you write a hook that returns JSX component, you are essentially defining the component within the functional component, so on each and every re-render you will be creating a new instance of the component. This will lead to the component being unmounted and mounted again. This is bad for performance and also buggy if you have stateful login within the component as the state will get reset with every re-render of the parent
* By defining a JSX component within the hook, you are taking away the option of lazy loading your component if the need be.
* Any performance optimization to the component will require you to make use of useMemo which doesn't give you the flexibility of a custom comparator function like React.memo()

The benefit on the other hand is that you have control over the state of the component in the parent. However, you can still implement the same logic by using a controlled component approach

*38*: **When would you want to avoid useEffect and use useLayoutEffect instead?**

If your effect is mutating the DOM (via a DOM node ref) ***and*** the DOM mutation will change the appearance of the DOM node between the time that it is rendered and your effect mutates it, then you **don't** want to use useEffect. You'll want to use useLayoutEffect. Otherwise, the user could see a flicker when your DOM mutations take effect. **This is pretty much the only time you want to avoid useEffect and use useLayoutEffect instead.**

React Hooks Code Challenges

*1***: Provide an example of any simple *Custom* React Hook. Why do we need Custom Hooks? Related To:** [**React**](https://www.fullstack.cafe/interview-questions/react)

A Custom Hook is a stateful function that uses other react built-in hooks (e.g. useState, useCallback etc.) that can wrap around the *stateful* logic that you wanted to gather in one place and *avoid copying and pasting the same logic* in multiple components.

Consider the increment/decriment custom hook:

| const useCounter = () => {  const [counter, setCounter] = useState(0);  return {  counter, // counter value  increment: () => setCounter(counter + 1), // function 1  decrement: () => setCounter(counter - 1) // function 2  };  }; |
| --- |

**and then in your component you can use it as follows:**

| const Component = () => {  const { counter, increment, decrement } = useCounter();  return (  <div>  <span onClick={decrement}>-</span>  <span style={{ padding: "10px" }}>{counter}</span>  <span onClick={increment}>+</span>  </div>  );  } |
| --- |

*2***: What is *equivalent* of this code using React Hooks? Related To:** [**React**](https://www.fullstack.cafe/interview-questions/react)

**Problem**

Let's say in our project we have componentWillUnmount that is used for cleanup (like removing event listeners, cancel the timer etc). How to refactor this code using React Hooks?

| componentDidMount() {  window.addEventListener('mousemove', () => {})  }  componentWillUnmount() {  window.removeEventListener('mousemove', () => {})  } |
| --- |

**Answer**

**React Hooks equivalent of above code will be as follows**

| useEffect(() => {  window.addEventListener('mousemove', () => {});  // returned function will be called on component unmount  return () => {  window.removeEventListener('mousemove', () => {})  }  }, []) |
| --- |

*3***: Can you initialise *state* from a function? Provide and example**

**Yes! Consider:**

| const StateFromFn = () => {  const [token] = useState(() => {  let token = window.localStorage.getItem("my-token");  return token || "default#-token#"  })  return <div>Token is {token}</div>  } |
| --- |

*4*: **How to mitigate multiple component re-renders when using multiple useState calls?**

**Problem**

| Consider this code:  const getData = url => {  const [data, setData] = useState(null);  const [loading, setLoading] = useState(true);  useEffect(async () => {  const test = await api.get('/people')  if(test.ok){  setLoading(false);  setData(test.data.results);  }  }, []);  return { data, loading };  }; |
| --- |

When I get data and update two different state variables (data and loading flag), my component (a data table) is rendered twice, even though both calls to the state updater are happening in the same function. Any ideas on how to mitigate this?

**Answer**

1. You could combine the loading state and data state into one state object and then you could do one setState call and there will only be one render.

Consider**:**

| const [userRequest, setUserRequest] = useState({  loading: false,  user: null,  });  useEffect(() => {  // Note that this replaces the entire object and deletes user key!  setUserRequest({ loading: true });  fetch('https://randomuser.me/api/')  .then(results => results.json())  .then(data => {  setUserRequest({  loading: false,  user: data.results[0],  });  });  }, []); |
| --- |

2 Before React 18, React will batch state updates if they're triggered from within a React-based event, like a button click or input change. It will not batch updates if they're triggered outside of a React event handler, like an async call.

3 Starting in React 18 (opt-in feature) all updates will be automatically batched, no matter where they originate from.

*5*: **Provide a good example of using useCallback hook in React**

Imagine you have a component that renders a big list of items. To prevent useless list re-renderings, you wrap it into React.memo().

| import useSearch from './fetch-items';  function MyBigList({ term, onItemClick }) {  const items = useSearch(term);  const map = item => <div onClick={onItemClick}>{item}</div>;  return <div>{items.map(map)}</div>;  }  export default React.memo(MyBigList); |
| --- |

The parent component of MyBigList provides a handler function to know when an item is clicked:

import { useCallback } from 'react';

| export function MyParent({ term }) {  const onItemClick = useCallback(event => {  console.log('You clicked ', event.currentTarget);  }, [term]);  return (  <MyBigList  term={term}  onItemClick={onItemClick}  />  );  } |
| --- |

onItemClick callback is memoized by useCallback(). As long as term is the same, useCallback() returns the *same function object* that passed to NyBigList as a prop. When MyParent component re-renders, onItemClick function object *remains the same* and **doesn't break the memoization** of MyBigList. That's the right use case to use useCallback.

*6***: Can we mimic componentWillMount() using React Hooks?**

**This is the way how I simulate constructor in functional components using the useRef hook.**

| export const useComponentWillMount = (func) => {  const willMount = useRef(true)  if (willMount.current) func()  willMount.current = false  }  // or  export const useComponentWillMount = (func) => {  useMemo(func, [])  } |
| --- |

**And usage:**

| const Component = (props) => {  useComponentWillMount(() => console.log("Runs only once before component mounts"));  ...  return (  <div>{...}</div>  );  } |
| --- |

*7* **: How would you *optimise* this code using one of the React Hooks?**

Consider this code. Imagine there are 100000 of users. Do you see any optimization we can make?

| import React from 'react';    const users = [  { id: 'a', name: 'Robin' },  { id: 'b', name: 'Dennis' },  ...  ];    const App = () => {  const [text, setText] = React.useState('');  const [search, setSearch] = React.useState('');    const handleText = (event) => {  setText(event.target.value);  };    const handleSearch = () => {  setSearch(text);  };    const filteredUsers = users.filter((user) => {  return user.name.toLowerCase().includes(search.toLowerCase());  });    return (  <div>  <input type="text" value={text} onChange={handleText} />  <button type="button" onClick={handleSearch}>  Search  </button>    <List list={filteredUsers} />  </div>  );  };    const List = ({ list }) => {  return (  <ul>  {list.map((item) => (  <ListItem key={item.id} item={item} />  ))}  </ul>  );  };    const ListItem = ({ item }) => {  return <li>{item.name}</li>;  };    export default App; |
| --- |

**Tip: try to find an use case for useMemo hook**

**Answer**

Even though the filteredUsers don't change when someone types into the input field, because they change only when clicking the button via the search state, the filter's callback function runs again and again for every keystroke in the input field.

| function App() {  ...    const filteredUsers = users.filter((user) => {  console.log('Filter function is running ...');  return user.name.toLowerCase().includes(search.toLowerCase());  });    ...  } |
| --- |

This doesn't slow down this small React application. However, if we would deal with a large set of data in this array and run the filter's callback function for every keystroke, we would maybe slow down the application.

you can use React's useMemo Hook to memoize a functions return value(s) and to run a function only if its dependencies (here search) have changed:

| function App() {  ...    const filteredUsers = React.useMemo(  () =>  users.filter((user) => {  console.log('Filter function is running ...');  return user.name.toLowerCase().includes(search.toLowerCase());  }),  [search]  );    ...  } |
| --- |

*8***: Explain the difference between useMemo vs useEffect + useState usage in that code**

**Consider:**

* useEffect + useState

| import { expensiveCalculation } from "foo";  function useCalculate(someNumber: number): number {  const [result, setResult] = useState<number>(null);  useEffect(() => {  setResult(expensiveCalculation(someNumber));  }, [someNumber]);  return result;  } |
| --- |

* useMemo

|  |
| --- |

import { expensiveCalculation } from "foo";

function useCalculateWithMemo(someNumber: number): number {

return useMemo(() => {

return expensiveCalculation(someNumber);

}, [someNumber]);

};

Are there any benefits in using useMemo (e.g. for an intensive function call) instead of using a combination of useEffect and useState?

**Answer**

The useEffect and setState will cause extra renders on every change: the first render will "lag behind" with stale data and then it'll immediately queue up an additional render with the new data.

Lets suppose x is initially 0:

* The useMemo version immediately renders 1.
* The useEffect version renders null, then after the component renders the effect runs, changes the state, and queues up a new render with 1.

Then if we change x to 2:

* The useMemo runs and 3 is rendered.
* The useEffect version runs, and renders 1 again, then the effect triggers and the component reruns with the correct value of 3.

In terms of how often expensiveCalculation runs, the two have identical behavior, but the useEffect version is causing twice as much rendering which is bad for performance.

Plus, the useMemo version is just cleaner and more readable,

**Redux**

*1*: **Do you need to keepIs all component states in Redux store?**

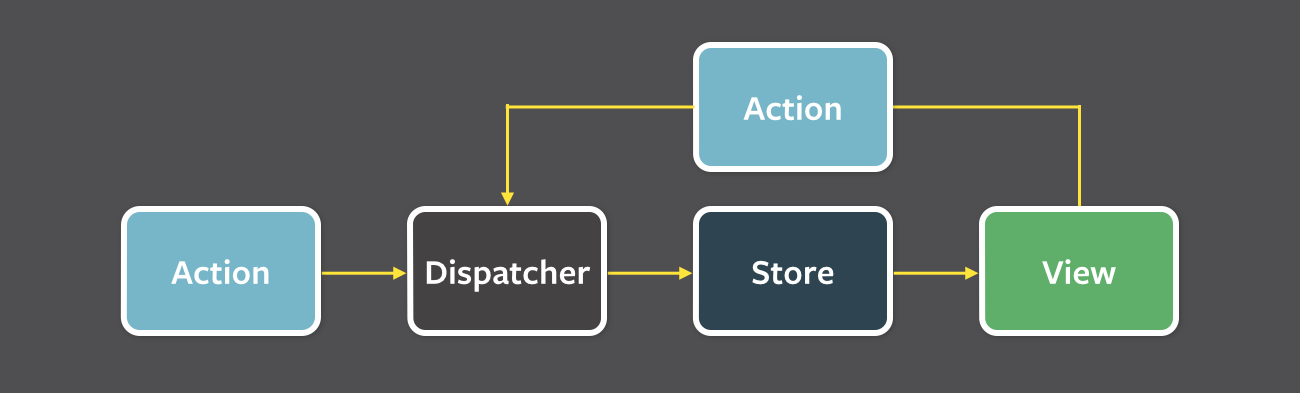
You need to keep your application state as small as possible. You don't have to push everything in there. Only do it makes a lot of sense to keep something there Or if it makes your life easier when using Dev Tools. But we shouldn't overload its importance too much.

*2*: **What is *Redux*?**

**Redux** is a predictable state container for JavaScript apps based on the Flux design pattern. Redux can be used together with ReactJS, or with any other view library. It is tiny (about 2kB) and has no dependencies.

*3*: **What is *Flux*?**

**Flux** is an application design paradigm used as a replacement for the more traditional mvc pattern. It is not a framework or a library but a new kind of architecture that complements React and the concept of Unidirectional Data Flow. Facebook used this pattern internally when working with React The workflow between dispatcher, stores and views components with distinct inputs and outputs as follows:



*4*: **What is Redux DevTools?**

**Redux DevTools** is a live-editing time travel environment for redux with hot reloading, action replay, and customizable UI. If you don’t want to bother with installing Redux DevTools and integrating it into your project, consider using Redux DevTools Extension for Chrome and Firefox.

*5*: **What is the difference between *Component* and *Container* in Redux?** Related To: [React](https://www.fullstack.cafe/interview-questions/react)

* **Component** is part of the React API. A Component is a class or function that describes part of a React UI.
* **Container** is an informal term for a React component that is connected to a redux store. Containers receive Redux state updates and dispatch actions, and they usually don't render DOM elements; they delegate rendering to presentational child components.

*6*: **What are reducers in redux?**

The **reducer** is a *pure function* that takes the previous state and an action, and returns the next state.

(previousState, action) => newState

It's called a reducer because it's the type of function you would pass to Array.prototype.reduce(reducer, ?initialValue). It's very important that the reducer stays *pure*. Things you should never do inside a reducer:

* Mutate its arguments;
* Perform side effects like API calls and routing transitions;
* Call non-pure functions, e.g. Date.now() or Math.random().

*7*: **What is redux-saga?**

**redux-saga** is a library that aims to make side effects (i.e. asynchronous things like data fetching and impure things like accessing the browser cache) in React/Redux applications easier and better. It is available in NPM as

npm install --save redux-saga

*8***: How to set initial state in Redux?**

**You need to pass initial state as second argument to createStore**

| const rootReducer = combineReducers({  todos: todos,  visibilityFilter: visibilityFilter  });  const initialState = {  todos: [{id:123, name:'sudheer', completed: false}]  };  const store = createStore(  rootReducer,  initialState  ); |
| --- |

*9***: What is the difference between React context and React redux?**

You can use **Context** in your application directly and is going to be great for passing down data to deeply nested components which what it was designed for. Whereas **Redux** is much more powerful and provides a large number of features that the Context Api doesn't provide.

Also, React Redux uses context internally but it doesn’t expose this fact in the public API. So you should feel much safer using context via React Redux than directly because if it changes, the burden of updating the code will be on React Redux instead developer responsibility.

*10*: **What is Redux Thunk?**

**Redux Thunk** middleware allows you to write action creators that return a function instead of an action. The thunk can be used to delay the dispatch of an action, or to dispatch only if a certain condition is met. The inner function receives the store methods dispatch and getState() as parameters.

*11*: **How to add multiple middlewares to Redux?**

You can use applyMiddleware where you can pass each piece of middleware as a new argument. So you just need to pass each piece of middleware you'd like.

For example, you can add Redux Thunk and logger middlewares as an argument as below,

| import { createStore, applyMiddleware } from 'redux'  const createStoreWithMiddleware = applyMiddleware(ReduxThunk, logger)(createStore); |
| --- |

*12*: **What are the features of Redux DevTools?**

Below are the major features of Redux devTools 1. Lets you inspect every state and action payload 2. Lets you go back in time by “cancelling” actions 3. If you change the reducer code, each “staged” action will be re-evaluated 4. If the reducers throw, you will see during which action this happened, and what the error was 5. With persistState() store enhancer, you can persist debug sessions across page reloads

*13*: **How to structure Redux top level directories?**

Most of the applications has several top-level directories as below 1. **Components** Used for “dumb” React components unaware of Redux 2. **Containers** Used for “smart” React components connected to Redux 3. **Actions** Used for all action creators, where file name corresponds to part of the app 4. **Reducers** Used for all reducers, where file name corresponds to state key 5. **Store** Used for store initialization This structure works well for small and mid-level size apps.

*14*: **What are Redux selectors and Why to use them?**

Selectors are functions that take Redux state as an argument and return some data to pass to the component.

For example, to get user details from the state:

| const getUserData = state => state.user.data; |
| --- |

*15*: **What are the core principles of Redux?**

Redux follows three fundamental principles:

1. **Single source of truth:** The state of your whole application is stored in an object tree within a single store. The single state tree makes it easier to keep track of changes over time and debug or inspect the application.
2. **State is ready only:** The only way to change the state is to emit an action, an object describing what happened. This ensures that neither the views nor the network callbacks will ever write directly to the state.
3. **Changes are made with pure functions:** To specify how the state tree is transformed by actions, you write pure reducers(Reducers are just pure functions that take the previous state and an action, and return the next state).

*16*: **What is a store in Redux?**

The *store* is a JavaScript object that holds application state. Along with this it also has the following responsibilities:

* 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).

*17*: **What are typical middleware choices for handling asynchronous calls in Redux?**

* Redux Thunk,
* Redux Promise,
* Redux Saga

*18*: **Are there any similarities between Redux and RxJS?**

These libraries are very different for very different purposes, but there are some vague similarities.

* **Redux** is a tool for managing state throughout the application. It is usually used as an architecture for UIs. Think of it as an alternative to (half of) Angular.
* **RxJS** is a reactive programming library. It is usually used as a tool to accomplish asynchronous tasks in JavaScript. Think of it as an alternative to Promises.

Redux uses the Reactive paradigm little bit because the Store is reactive. The Store observes actions from a distance, and changes itself. RxJS also uses the Reactive paradigm, but instead of being an architecture, it gives you basic building blocks, Observables, to accomplish this "observing from a distance" pattern.

*19*: **How to access redux store outside a react component?**

Yes.You just need to export the store from the module where it created with createStore. Also, it shouldn't pollute the global window object

| store = createStore(myReducer);  export default store; |
| --- |

*20*: **What is Redux form?**

**Redux Form** works with React and Redux to enable a form in React to use Redux to store all of its state. Redux Form can be used with raw HTML5 inputs, but it also works with very well with common UI frameworks like Material UI, React Widgets and React Bootstrap.

*21*: **What are the main features of Redux Form?**

Below are the major features of redux form: 1. Field values persistence via Redux store 2. Validation (sync/async) and submission 3. Formatting, parsing and normalization of field values

*22*: **What are the downsides of Redux over Flux?**

Instead of saying downsides we can say that there are few compromises of using Redux over Flux. Those are as follows: 1. **You will need to learn avoiding mutations:** Flux is un-opinionated about mutating data, but Redux doesn't like mutations and many packages complementary to Redux assume you never mutate the state. You can enforce this with dev-only packages like redux-immutable-state-invariant, Immutable.js, or your team to write non-mutative code. 2. **You're going to have to carefully pick your packages:** While Flux explicitly doesn't try to solve problems such as undo/redo, persistence, or forms, Redux has extension points such as middleware and store enhancers, and it has spawned a young but rich ecosystem. This means most packages are new ideas and haven't received the critical mass of usage yet 3. **There is no nice Flow integration yet:** Flux currently lets you do very impressive static type checks which Redux doesn't support yet.

2*3*: **How to use connect from react redux?**

You need to follow two steps to use your store in your container

1. **Use mapStateToProps():** It maps the state variables from your store to the props that you specify 2. **Connect the above props to your container:** The object returned by the mapStateToProps component is connected to the container.

You can import connect from react-redux like

| import React from 'react';  import { connect } from 'react-redux';  class App extends React.Component {  render() {  return <div>{this.props.containerData}</div>;  }  }  function mapStateToProps(state) {  return { containerData: state.appData };  }  export default connect(mapStateToProps)(App);  function mapStateToProps(state) {  return { containerData: state.data };  }  export default connect(mapStateToProps)(App); |
| --- |

*24*: **What is the purpose of the constants in Redux?**

Constants allow you to easily find all usages of that specific functionality across the project when you use an IDE. It also prevents you from introducing silly bugs caused by typos -- in which case, you will get a ReferenceError immediately.

1 Normally we will save them in a single file (constants.js or actionTypes.js) For example:

| export const ADD\_TODO = 'ADD\_TODO';  export const DELETE\_TODO = 'DELETE\_TODO';  export const EDIT\_TODO = 'EDIT\_TODO';  export const COMPLETE\_TODO = 'COMPLETE\_TODO';  export const COMPLETE\_ALL = 'COMPLETE\_ALL';  export const CLEAR\_COMPLETED = 'CLEAR\_COMPLETED'; |
| --- |

In redux you use them in two places 1. **During actions creation** Let's take actions.js

| import { ADD\_TODO } from './actionTypes';  export function addTodo(text) {  return { type: ADD\_TODO, text };  } |
| --- |

**2 Reducers** Let's create reducer.js

| import { ADD\_TODO } from './actionTypes';  export default (state = [], action) => {  switch (action.type) {  case ADD\_TODO:  return [  ...state,  {  text: action.text,  completed: false  }  ];  default:  return state  }  }; |
| --- |

*25*: **What are the differences between redux-saga and redux-thunk?**

Both **Redux Thunk** and **Redux Saga** take care of dealing with side effects. In most of the scenarios, Thunk allows *Promises* to deal with them, whereas Saga uses *Generators*.

Thunk is simple to use and Promises are familiar to many developers, Saga/Generators are more powerful but you will need to learn them. But both the two middleware can coexists, so you can start with Thunks and introduce Sagas when/if you need them.

*26*: **What is a *Reducer*?**

A *reducer* is simply a pure function that takes the previous state and an action, and returns the next state.

*27*: **How to reset state in redux?**

You need to write a root reducer in your application which delegate handling the action to the reducer generated by combineReducers(). For example, let us take rootReducer to return the initial state after USER\_LOGOUT action. As we know, reducers are supposed to return the initial state when they are called with undefined as the first argument, no matter the action.

| const appReducer = combineReducers({  /\* your app’s top-level reducers \*/  })  const rootReducer = (state, action) => {  if (action.type === 'USER\_LOGOUT') {  state = undefined  }  return appReducer(state, action)  } |
| --- |

In case of using redux-persist, you may also need to clean your storage. Redux-perist keeps a copy of your state in a storage engine. First, you need to import the appropriate storage engine and then, to parse the state before setting it to undefined and clean each storage state key,

| const appReducer = combineReducers({  /\* your app’s top-level reducers \*/  })  const rootReducer = (state, action) => {  if (action.type === 'USER\_LOGOUT') {  Object.keys(state).forEach(key => {  storage.removeItem(`persist:${key}`);  });  state = undefined;  }  return appReducer(state, action)  } |
| --- |

*28***: How to make Ajax request in Redux?**

You can use redux-thunk middleware which allows you to define async actions. Let's take an example of fetching specific account as a ajax call using fetch API,

| export function fetchAccount(id) {  return dispatch => {  dispatch(setLoadingAccountState()); // Show a loading spinner  fetch(`/account/${id}`, (response) => {  dispatch(doneFetchingAccount()); // Hide loading spinner  if (response.status == 200){  dispatch(setAccount(response.json)); // Use a normal function to set the received state  } else {  dispatch(someError)  }  })  }  }  function setAccount(data) {  return { type: 'SET\_Account', data: data };  } |
| --- |

*29***: Whats the purpose of at (@) symbol in the redux connect decorator?**

The @ symbol is in fact a JavaScript expression used to signify decorators. Decorators make it possible to annotate and modify classes and properties at design time. Let's take an example setting up Redux without and with a decorator

**Without decorator**

| import React from 'react';  import \* as actionCreators from './actionCreators';  import { bindActionCreators } from 'redux';  import { connect } from 'react-redux';  function mapStateToProps(state) {  return { todos: state.todos };  }  function mapDispatchToProps(dispatch) {  return { actions: bindActionCreators(actionCreators, dispatch) };  }  class MyApp extends React.Component {  // ...define your main app here  }  export default connect(mapStateToProps, mapDispatchToProps)(MyApp); |
| --- |

**With decorator**

| import React from 'react';  import \* as actionCreators from './actionCreators';  import { bindActionCreators } from 'redux';  import { connect } from 'react-redux';  function mapStateToProps(state) {  return { todos: state.todos };  }  function mapDispatchToProps(dispatch) {  return { actions: bindActionCreators(actionCreators, dispatch) };  }  @connect(mapStateToProps, mapDispatchToProps)  export default class MyApp extends React.Component {  // ...define your main app here  } |
| --- |

The above examples are almost similar except the usage of decorator. The decorator syntax isn't built into any Javascript runtimes yet, and is still experimental and subject to change. You can use babel for the decorators support.

*30***: What are the differences between call and put in redux-saga?**

**Both call and put are effects creators functions.**

* **call** function is used to create effect description, which instructs middleware to call the promise.
* **put** function creates effect, in which instructs middleware to dispatch an action to the store.

**Let's take example of how these effects work for fetching particular user data**

| function\* fetchUserSaga(action) {  // `call` function accepts rest arguments, which will be passed to `api.fetchUser` function.  // Instructing middleware to call promise, it resolved value will be assigned to `userData` variable  const userData = yield call(api.fetchUser, action.userId);  // Instructing middleware to dispatch corresponding action.  yield put({  type: 'FETCH\_USER\_SUCCESS',  userData  });  } |
| --- |

*31***: Why are Redux state functions called as reducers?**

**Reducers** always return the accumulation of the state (based on all previous and current actions) not only default values. Therefore, they act as a reducer of state. Each time a redux reducer is called, the state is passed in with the action (state, action). This state is then reduced (or accumulated) based on the action, and then the next state is returned. i.e, You could "reduce" a collection of actions and an initial state (of the store) on which to perform these actions to get the resulting final state.

*32*: **What is the proper way to access Redux store?**

The best way to access your store through a component is using the connect() function. Actually creates a new component that wraps around your existing one! This pattern is called *Higher-Order Components*, and is generally the preferred way of extending a component's functionality in React.

This allows you to map state and action creators to your component, and have them passed in automatically as your store updates. Let's take an example of FilterLink component using connect,

| import { connect } from 'react-redux'  import { setVisibilityFilter } from '../actions'  import Link from '../components/Link'  const mapStateToProps = (state, ownProps) => {  return {  active: ownProps.filter === state.visibilityFilter  }  }  const mapDispatchToProps = (dispatch, ownProps) => {  return {  onClick: () => {  dispatch(setVisibilityFilter(ownProps.filter))  }  }  }  const FilterLink = connect(  mapStateToProps,  mapDispatchToProps  )(Link)  export default FilterLink |
| --- |

Due to it having quite a few performance optimizations and generally being less likely to cause bugs, the Redux devs almost always recommend using connect over accessing the store directly (using context API)

| class MyComponent {  someMethod() {  doSomethingWith(this.context.store);  }  } |
| --- |

*33*: **What is *Redux Thunk* used for?**

*Redux thunk* is middleware that allows you 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 you to handle the asyncronous dispatching of actions.

*34*: **What is the mental model of redux-saga?**

The mental model is that a saga is like a separate thread in your application that's solely responsible for side effects. redux-saga is a redux middleware, which means this thread can be started, paused and cancelled from the main application with normal redux actions, it has access to the full redux application state and it can dispatch redux actions as well.

*35*: **How Relay is different from Redux?**

**Relay** is similar to redux in that they both use a single store. The main difference is that relay only manages state originated from the server, and all access to the state is used via GraphQL queries (for reading data) and mutations (for changing data). Relay caches the data for you and optimizes data fetching for you, by fetching only changed data and nothing more.