**Core React**

1. **What is React?**

React is an **open-source frontend JavaScript library** which is used for building user interfaces especially for single page applications. It is used for handling view layer for web and mobile apps. React was created by Jordan Walke, a software engineer working for Facebook. React was first deployed on Facebook's News Feed in 2011 and on Instagram in 2012.

1. **What are the major features of React?**

The major features of React are:

* + It uses **VirtualDOM** instead of 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.

1. **What is JSX?**

*JSX* is a XML-like syntax extension to ECMAScript (the acronym stands for *JavaScript XML*). Basically it just provides syntactic sugar for the React.createElement() function, giving us expressiveness of JavaScript along with HTML like template syntax.

In the example below text inside <h1> tag is returned as JavaScript function to the render function.

class App extends React.Component {

render() {

return(

<div>

<h1>{'Welcome to React world!'}</h1>

</div>

)

}

}

1. **What is the difference between Element and Component?**

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 React.createElement() function returns an object:

{

type: 'div',

props: {

children: 'Login',

id: 'login-btn'

}

}

And finally it renders to the DOM using ReactDOM.render():

<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 or it can be defined as a function. In either case, it takes props as an input, and returns a JSX tree as the output:

const Button = ({ onLogin }) =>

<div id={'login-btn'} onClick={onLogin}>Login</div>

Then JSX gets transpiled to a React.createElement() function tree:

const Button = ({ onLogin }) => React.createElement(

'div',

{ id: 'login-btn', onClick: onLogin },

'Login'

)

1. **How to create components in React?**

There are two possible ways to create a component.

* + **Function Components:** This is the simplest way to create a component. Those are pure JavaScript functions that accept props object as first parameter and return React elements:
  + function Greeting({ message }) {
  + return <h1>{`Hello, ${message}`}</h1>

}

* + **Class Components:** You can also use ES6 class to define a component. The above function component can be written as:
  + class Greeting extends React.Component {
  + render() {
  + return <h1>{`Hello, ${this.props.message}`}</h1>
  + }

}

1. **When to use a Class Component over a Function Component?**

If the component needs *state or lifecycle methods* then use class component otherwise use function component. *However, from React 16.8 with the addition of Hooks, you could use state , lifecycle methods and other features that were only available in class component right in your function component.*

1. **What are Pure Components?**

*React.PureComponent* is exactly the same as *React.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.

1. **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.

Let's create a user component with message state,

class User extends React.Component {

constructor(props) {

super(props)

this.state = {

message: 'Welcome to React world'

}

}

render() {

return (

<div>

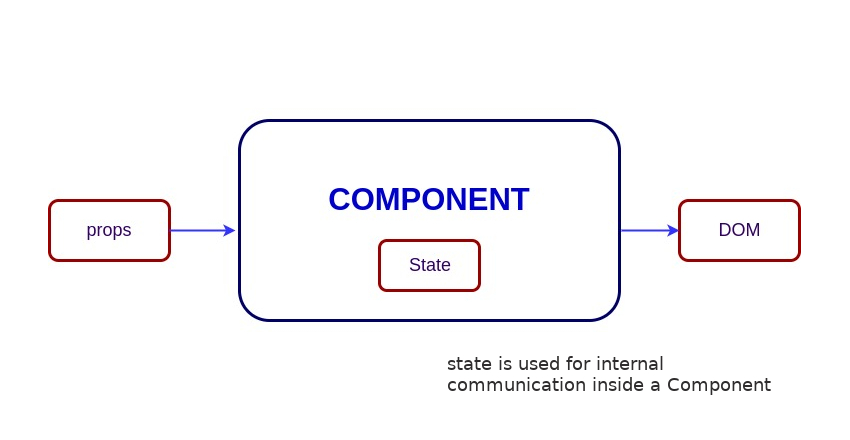
<h1>{this.state.message}</h1>

</div>

)

}

}

[](https://github.com/sudheerj/reactjs-interview-questions/blob/master/images/state.jpg)

State is similar to props, but it is private and fully controlled by the component. i.e, It is not accessible to any other component til the owner component decides to pass it.

1. **What are props in React?**

*Props* are inputs to components. They are single values or objects containing a set of values that are passed to components on creation using a naming convention similar to HTML-tag attributes. 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:

* + Pass custom data to your component.
  + Trigger state changes.
  + 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

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

1. **Why should we not update the 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:** You can directly assign to the state object either in *constructor* or using latest javascript's class field declaration syntax.

1. **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 than this callback function.

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

1. **What is the difference between HTML and React event handling?**

Below are some of the main differences between HTML and React event handling,

* 1. In HTML, the event name usually represents in *lowercase* as a convention:

<button onclick='activateLasers()'>

Whereas in React it follows *camelCase* convention:

<button onClick={activateLasers}>

* 1. In HTML, you can return false to prevent default behavior:

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

Whereas in React you must call preventDefault() explicitly:

function handleClick(event) {

event.preventDefault()

console.log('The link was clicked.')

}

* 1. In HTML, you need to invoke the function by appending () Whereas in react you should not append () with the function name. (refer "activateLasers" function in the first point for example)

1. **How to bind methods or event handlers in JSX callbacks?**

There are 3 possible ways to achieve this:

* 1. **Binding in Constructor:** In JavaScript classes, the methods are not bound by default. The same thing applies for React event handlers defined as class methods. Normally we bind them in constructor.
  2. class Component extends React.Component {
  3. constructor(props) {
  4. super(props)
  5. this.handleClick = this.handleClick.bind(this)
  6. }
  7. handleClick() {
  8. // ...
  9. }

}

* 1. **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.
  2. handleClick = () => {
  3. console.log('this is:', this)

}

<button onClick={this.handleClick}>

{'Click me'}

</button>

* 1. **Arrow functions in callbacks:** You can use *arrow functions* directly in the callbacks.
  2. <button onClick={(event) => this.handleClick(event)}>
  3. {'Click me'}

</button>

**Note:** If the callback is passed as prop to child components, those components might do an extra re-rendering. In those cases, it is preferred to go with .bind() or *public class fields syntax* approach considering performance.

1. **How to pass a parameter to an event handler or callback?**

You can use an *arrow function* to wrap around an *event handler* and pass parameters:

<button onClick={() => this.handleClick(id)} />

This is an equivalent to calling .bind:

<button onClick={this.handleClick.bind(this, id)} />

Apart from these two approaches, you can also pass arguments to a function which is defined as arrow function

<button onClick={this.handleClick(id)} />

handleClick = (id) => () => {

console.log("Hello, your ticket number is", id)

};

1. **What are synthetic events in React?**

SyntheticEvent is a cross-browser wrapper around the browser's native event. It's API is same as the browser's native event, including stopPropagation() and preventDefault(), except the events work identically across all browsers.

1. **What are inline conditional expressions?**

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 &&.

<h1>Hello!</h1>

{

messages.length > 0 && !isLogin?

<h2>

You have {messages.length} unread messages.

</h2>

:

<h2>

You don't have unread messages.

</h2>

}

1. **What is "key" prop and what is the benefit of using it in arrays of elements?**

A key is a special string attribute you **should** include when creating arrays of elements. *Key* prop helps React identify which items have changed, are added, or are removed.

Most often we use ID from our data as *key*:

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. Using *indexes* for *keys* is **not recommended** 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 of li tag.
  3. There will be a warning message in the console if the key prop is not present on list items.

1. **What is the use of refs?**

The *ref* is used to return a reference to the element. They *should be avoided* in most cases, however, they can be useful when you need a direct access to the DOM element or an instance of a component.

1. **How to create refs?**

There are two approaches

* 1. This is a recently added approach. *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 within constructor.
  2. class MyComponent extends React.Component {
  3. constructor(props) {
  4. super(props)
  5. this.myRef = React.createRef()
  6. }
  7. render() {
  8. return <div ref={this.myRef} />
  9. }

}

* 1. You can also use ref callbacks approach regardless of React version. For example, the search bar component's input element accessed as follows,
  2. class SearchBar extends Component {
  3. constructor(props) {
  4. super(props);
  5. this.txtSearch = null;
  6. this.state = { term: '' };
  7. this.setInputSearchRef = e => {
  8. this.txtSearch = e;
  9. }
  10. }
  11. onInputChange(event) {
  12. this.setState({ term: this.txtSearch.value });
  13. }
  14. render() {
  15. return (
  16. <input
  17. value={this.state.term}
  18. onChange={this.onInputChange.bind(this)}
  19. ref={this.setInputSearchRef} />
  20. );
  21. }

}

You can also use *refs* in function components using **closures**. **Note**: You can also use inline ref callbacks even though it is not a recommended approach

1. **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>

1. **Which is preferred option with in callback refs and findDOMNode()?**

It is preferred to use *callback refs* over findDOMNode() API. Because findDOMNode() prevents certain improvements in React in the future.

The **legacy** approach of using findDOMNode:

class MyComponent extends Component {

componentDidMount() {

findDOMNode(this).scrollIntoView()

}

render() {

return <div />

}

}

The recommended approach is:

class MyComponent extends Component {

constructor(props){

super(props);

this.node = createRef();

}

componentDidMount() {

this.node.current.scrollIntoView();

}

render() {

return <div ref={this.node} />

}

}

1. **Why are String Refs legacy?**

If you worked with React before, you might be familiar with an older API where the ref attribute is a string, like ref={'textInput'}, and the DOM node is accessed as this.refs.textInput. We advise against it because *string refs have below issues*, and are considered legacy. String refs were **removed in React v16**.

* 1. They *force React to keep track of currently executing component*. This is problematic because it makes react module stateful, and thus causes weird errors when react module is duplicated in the bundle.
  2. They are *not composable* — if a library puts a ref on the passed child, the user can't put another ref on it. Callback refs are perfectly composable.
  3. They *don't work with static analysis* like Flow. Flow can't guess the magic that framework does to make the string ref appear on this.refs, as well as its type (which could be different). Callback refs are friendlier to static analysis.
  4. It doesn't work as most people would expect with the "render callback" pattern (e.g. )
  5. class MyComponent extends Component {
  6. renderRow = (index) => {
  7. // This won't work. Ref will get attached to DataTable rather than MyComponent:
  8. return <input ref={'input-' + index} />;
  9. // This would work though! Callback refs are awesome.
  10. return <input ref={input => this['input-' + index] = input} />;
  11. }
  12. render() {
  13. return <DataTable data={this.props.data} renderRow={this.renderRow} />
  14. }

}

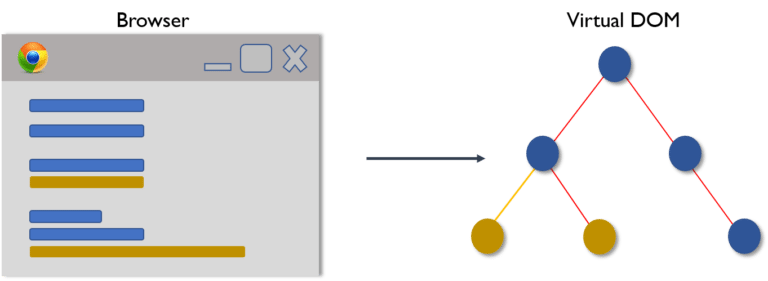
1. **What is Virtual DOM?**

The *Virtual DOM* (VDOM) is an in-memory representation of *Real DOM*. The representation of a UI is kept in memory and synced with the "real" DOM. It's a step that happens between the render function being called and the displaying of elements on the screen. This entire process is called *reconciliation*.

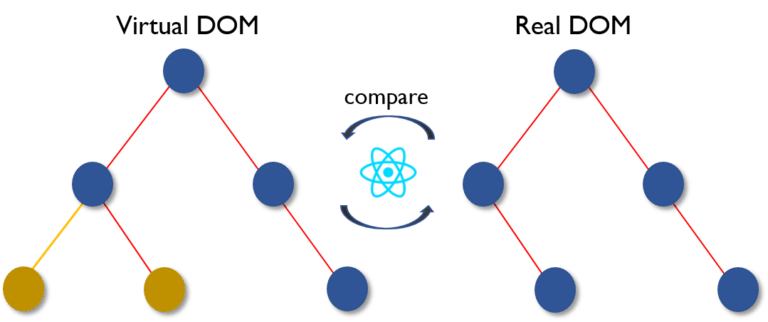
1. **How Virtual DOM works?**

The *Virtual DOM* works in three simple steps.

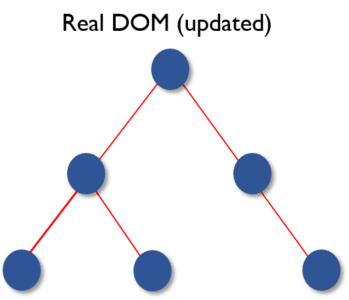
* 1. Whenever any underlying data changes, the entire UI is re-rendered in Virtual DOM representation.

[](https://github.com/sudheerj/reactjs-interview-questions/blob/master/images/vdom1.png)

* 1. Then the difference between the previous DOM representation and the new one is calculated.

[](https://github.com/sudheerj/reactjs-interview-questions/blob/master/images/vdom2.png)

* 1. Once the calculations are done, the real DOM will be updated with only the things that have actually changed.

[](https://github.com/sudheerj/reactjs-interview-questions/blob/master/images/vdom3.png)

1. **What is the difference between Shadow DOM and Virtual DOM?**

The *Shadow DOM* is a browser technology designed primarily for scoping variables and CSS in *web components*. The *Virtual DOM* is a concept implemented by libraries in JavaScript on top of browser APIs.

1. **What is React Fiber?**

Fiber is the new *reconciliation* engine or reimplementation of core algorithm in React v16. The goal of React Fiber is to increase its suitability for areas like animation, layout, gestures, ability to pause, abort, or reuse work and assign priority to different types of updates; and new concurrency primitives.

1. **What is the main goal of React Fiber?**

The goal of *React Fiber* is to increase its suitability for areas like animation, layout, and gestures. Its headline feature is **incremental rendering**: the ability to split rendering work into chunks and spread it out over multiple frames.

*from documentation*

Its main goals are:

* 1. Ability to split interruptible work in chunks.
  2. Ability to prioritize, rebase and reuse work in progress.
  3. Ability to yield back and forth between parents and children to support layout in React.
  4. Ability to return multiple elements from render().
  5. Better support for error boundaries.

1. **What are controlled components?**

A component that controls the input elements within the forms on subsequent user input is called **Controlled Component**, i.e, every state mutation will have an associated handler function.

For example, to write all the names in uppercase letters, we use handleChange as below,

handleChange(event) {

this.setState({value: event.target.value.toUpperCase()})

}

1. **What are uncontrolled components?**

The **Uncontrolled Components** are the ones that store their own state internally, and you query the DOM using a ref to find its current value when you need it. This is a bit more like traditional HTML.

In the below UserProfile component, the name input is accessed using ref.

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's recommend to use controlled components to implement forms.

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

JSX elements will be transpiled to React.createElement() functions to create React elements which are going to be used for the object representation of UI. Whereas cloneElement is used to clone an element and pass it new props.

1. **What is Lifting State Up in React?**

When several components need to share the same changing data then it is recommended to *lift the shared state up* to their closest common ancestor. That means if two child components share the same data from its parent, then move the state to parent instead of maintaining local state in both of the child components.

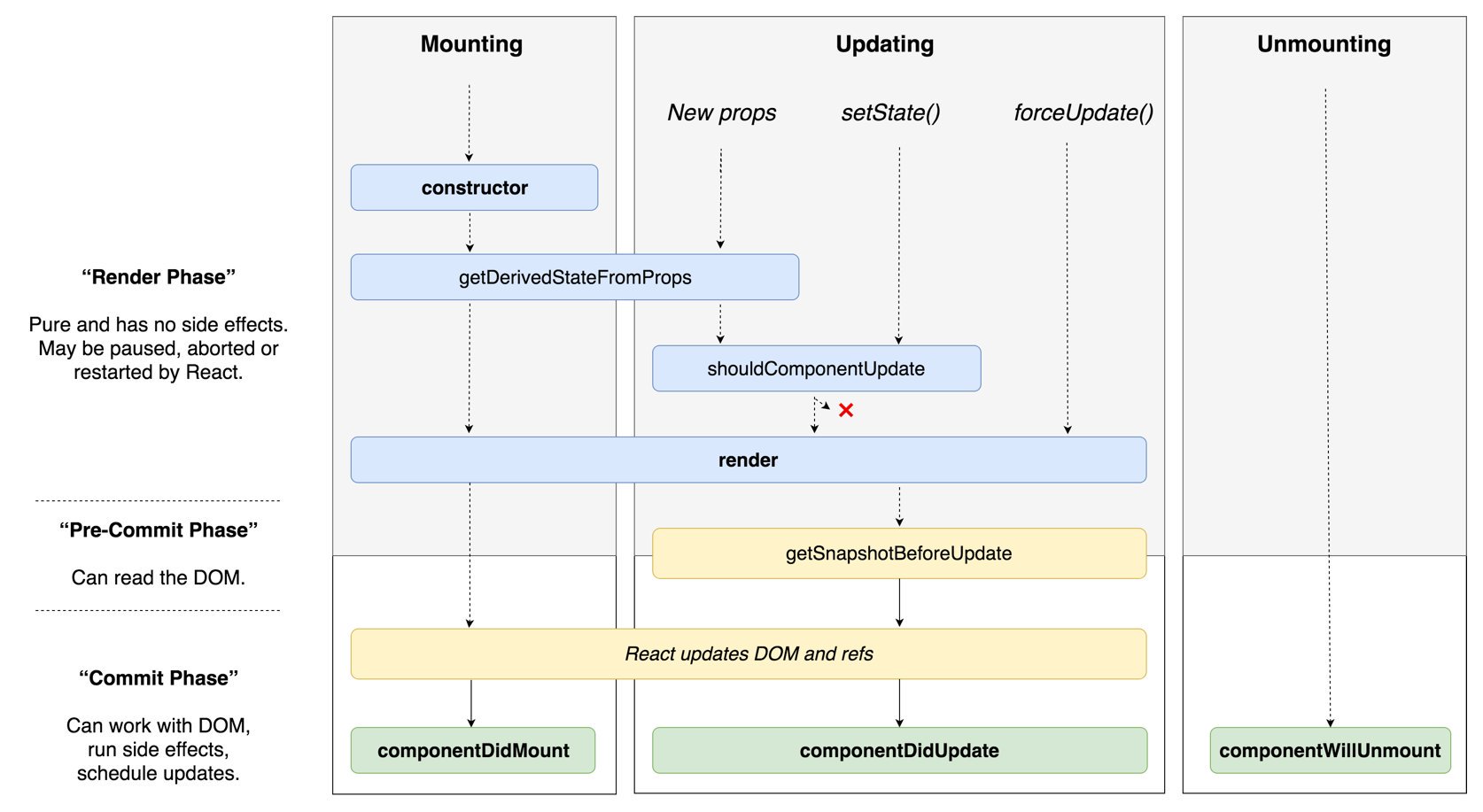
1. **What are the different phases of component lifecycle?**

The component lifecycle has three distinct lifecycle phases:

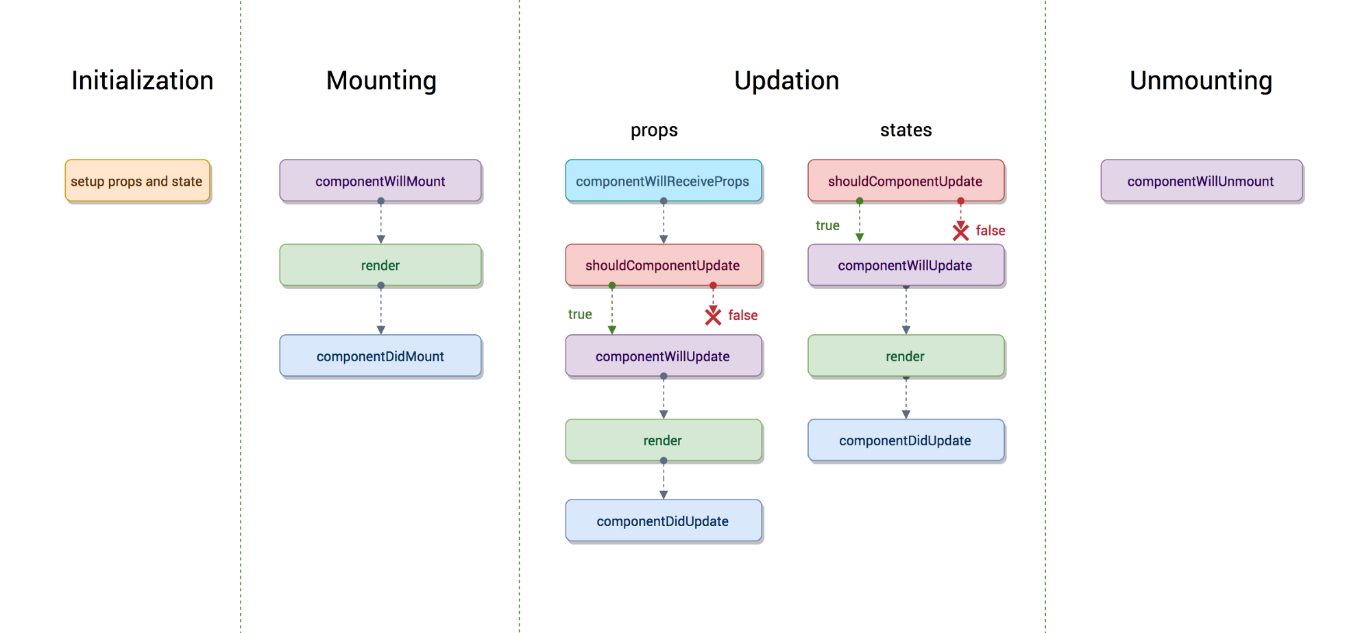
* 1. **Mounting:** The component is ready to mount in the browser DOM. This phase covers initialization from constructor(), getDerivedStateFromProps(), render(), and componentDidMount() lifecycle methods.
  2. **Updating:** In this phase, the component get updated in two ways, sending the new props and updating the state either from setState() or forceUpdate(). This phase covers getDerivedStateFromProps(), shouldComponentUpdate(), render(), getSnapshotBeforeUpdate() and componentDidUpdate() lifecycle methods.
  3. **Unmounting:** In this last phase, the component is not needed and get unmounted from the browser DOM. This phase includes componentWillUnmount() lifecycle method.

It's worth mentioning that React internally has a concept of phases when applying changes to the DOM. They are separated as follows

* 1. **Render** The component will render without any side-effects. This applies for Pure components and in this phase, React can pause, abort, or restart the render.
  2. **Pre-commit** Before the component actually applies the changes to the DOM, there is a moment that allows React to read from the DOM through the getSnapshotBeforeUpdate().
  3. **Commit** React works with the DOM and executes the final lifecycles respectively componentDidMount() for mounting, componentDidUpdate() for updating, and componentWillUnmount() for unmounting.

[](https://github.com/sudheerj/reactjs-interview-questions/blob/master/images/phases16.3.jpg)

Before React 16.3

[](https://github.com/sudheerj/reactjs-interview-questions/blob/master/images/phases.png)

1. **What are the lifecycle methods of React?**

Before React 16.3

* + **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 event listeners 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 re-render if component receives new prop.
  + **componentWillUpdate:** Executed before re-rendering the component when there are props & 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.

React 16.3+

* + **getDerivedStateFromProps:** Invoked right before calling render() and is invoked on *every* render. This exists for rare use cases where you need derived state. Worth reading if you need derived state.
  + **componentDidMount:** Executed after first rendering and where all AJAX requests, DOM or state updates, and set up event listeners should occur.
  + **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 re-render if component receives new prop.
  + **getSnapshotBeforeUpdate:** Executed right before rendered output is committed to the DOM. Any value returned by this will be passed into componentDidUpdate(). This is useful to capture information from the DOM i.e. scroll position.
  + **componentDidUpdate:** Mostly it is used to update the DOM in response to prop or state changes. This will not fire if shouldComponentUpdate() returns false.
  + **componentWillUnmount** It will be used to cancel any outgoing network requests, or remove all event listeners associated with the component.

1. **What are Higher-Order Components?**

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 **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:

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

1. **How to create props proxy for HOC component?**

You can add/edit props passed to the component using *props proxy* pattern like this:

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} />

}

}

}

1. **What is context?**

*Context* provides a way to pass data through the component tree without having to pass props down manually at every level.

For example, authenticated user, locale preference, UI theme need to be accessed in the application by many components.

const {Provider, Consumer} = React.createContext(defaultValue)

1. **What is children prop?**

*Children* is a prop (this.props.children) that allow you to pass components as data to other components, just like any other prop you use. Component tree put between component's opening and closing tag will be passed to that component as children prop.

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,

const MyDiv = React.createClass({

render: function() {

return <div>{this.props.children}</div>

}

})

ReactDOM.render(

<MyDiv>

<span>{'Hello'}</span>

<span>{'World'}</span>

</MyDiv>,

node

)

1. **How to write comments in React?**

The comments in React/JSX are similar to JavaScript Multiline comments but are wrapped in curly braces.

**Single-line comments:**

<div>

{/\* Single-line comments(In vanilla JavaScript, the single-line comments are represented by double slash(//)) \*/}

{`Welcome ${user}, let's play React`}

</div>

**Multi-line comments:**

<div>

{/\* Multi-line comments for more than

one line \*/}

{`Welcome ${user}, let's play React`}

</div>

1. **What is the purpose of using super constructor with props argument?**

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: 'John', age: 42 }

}

}

**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: 'John', age: 42 }

}

render() {

// no difference outside constructor

console.log(this.props) // prints { name: 'John', age: 42 }

}

}

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

1. **What is reconciliation?**

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*.

1. **How to set state with a dynamic key name?**

If you are using ES6 or the Babel transpiler to transform your JSX code then you can accomplish this with *computed property names*.

handleInputChange(event) {

this.setState({ [event.target.id]: event.target.value })

}

1. **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: 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: handleClick is passed as a reference!

return <button onClick={this.handleClick}>{'Click Me'}</button>

}

1. **Is lazy function supports named exports?**

No, currently React.lazy function supports default exports only. If you would like to import modules which are named exports, you can create an intermediate module that reexports it as the default. It also ensures that tree shaking keeps working and don’t pull unused components. Let's take a component file which exports multiple named components,

// MoreComponents.js

export const SomeComponent = /\* ... \*/;

export const UnusedComponent = /\* ... \*/;

and reexport MoreComponents.js components in an intermediate file IntermediateComponent.js

// IntermediateComponent.js

export { SomeComponent as default } from "./MoreComponents.js";

Now you can import the module using lazy function as below,

import React, { lazy } from 'react';

const SomeComponent = lazy(() => import("./IntermediateComponent.js"));

1. **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. Pass a string as the className prop.

render() {

return <span className={'menu navigation-menu'}>{'Menu'}</span>

}

1. **What are fragments?**

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*, but it's not supported in many tools:

render() {

return (

<>

<ChildA />

<ChildB />

<ChildC />

</>

)

}

1. **Why fragments are better than container divs?**

Below are the list of reasons,

* 1. Fragments are a bit faster and use less memory by not creating an extra DOM node. This only has a real benefit on very large and deep trees.
  2. Some CSS mechanisms like *Flexbox* and *CSS Grid* have a special parent-child relationships, and adding divs in the middle makes it hard to keep the desired layout.
  3. The DOM Inspector is less cluttered.

1. **What are portals in React?**

*Portal* is a recommended way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.

ReactDOM.createPortal(child, container)

The first argument is any render-able React child, such as an element, string, or fragment. The second argument is a DOM element.

1. **What are stateless components?**

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 function components. There are a lot of benefits if you decide to use function components here; they are easy to write, understand, and test, a little faster, and you can avoid the this keyword altogether.

1. **What are stateful components?**

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() {

// ...

}

}

**React 16.8 Update:**

Hooks let you use state and other React features without writing classes.

*The Equivalent Functional Component*

import React, {useState} from 'react';

const App = (props) => {

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

return (

// JSX

)

}

1. **How to apply validation on props in React?**

When the application is running in *development mode*, React will automatically check all props that we set on components to make sure they have *correct type*. If the type is incorrect, React will generate warning messages in the console. It's disabled in *production mode* due to performance impact. The mandatory props are defined with isRequired.

The set of predefined prop types:

* 1. PropTypes.number
  2. PropTypes.string
  3. PropTypes.array
  4. PropTypes.object
  5. PropTypes.func
  6. PropTypes.node
  7. PropTypes.element
  8. PropTypes.bool
  9. PropTypes.symbol
  10. PropTypes.any

We can define propTypes for User component as below:

import React from 'react'

import PropTypes from 'prop-types'

class User extends React.Component {

static propTypes = {

name: PropTypes.string.isRequired,

age: PropTypes.number.isRequired

}

render() {

return (

<>

<h1>{`Welcome, ${this.props.name}`}</h1>

<h2>{`Age, ${this.props.age}`}</h2>

</>

)

}

}

**Note:** In React v15.5 *PropTypes* were moved from React.PropTypes to prop-types library.

*The Equivalent Functional Component*

import React from 'react'

import PropTypes from 'prop-types'

function User() {

return (

<>

<h1>{`Welcome, ${this.props.name}`}</h1>

<h2>{`Age, ${this.props.age}`}</h2>

</>

)

}

User.propTypes = {

name: PropTypes.string.isRequired,

age: PropTypes.number.isRequired

}

1. **What are the advantages of React?**

Below are the list of main advantages of React,

* 1. Increases the application's performance with *Virtual DOM*.
  2. JSX makes code easy to read and write.
  3. It renders both on client and server side (*SSR*).
  4. Easy to integrate with frameworks (Angular, Backbone) since it is only a view library.
  5. Easy to write unit and integration tests with tools such as Jest.

1. **What are the limitations of React?**

Apart from the advantages, there are few limitations of React too,

* 1. React is just a view library, not a full framework.
  2. There is a learning curve for beginners who are new to web development.
  3. Integrating React 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.

1. **What are error boundaries in React v16?**

*Error boundaries* are 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) or static getDerivedStateFromError() :

class ErrorBoundary extends React.Component {

constructor(props) {

super(props)

this.state = { hasError: false }

}

componentDidCatch(error, info) {

// You can also log the error to an error reporting service

logErrorToMyService(error, info)

}

static getDerivedStateFromError(error) {

// Update state so the next render will show the fallback UI.

return { hasError: true };

}

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>

1. **How error boundaries handled in React v15?**

React v15 provided very basic support for *error boundaries* using unstable\_handleError method. It has been renamed to componentDidCatch in React v16.

1. **What are the recommended ways for static type checking?**

Normally we use *PropTypes library* (React.PropTypes moved to a prop-types package since React v15.5) for *type checking* in the React applications. For large code bases, it is recommended to use *static type checkers* such as Flow or TypeScript, that perform type checking at compile time and provide auto-completion features.

1. **What is the use of react-dom package?**

The react-dom package provides *DOM-specific methods* that can be used at the top level of your app. Most of the components are not required to use this module. Some of the methods of this package are:

* 1. render()
  2. hydrate()
  3. unmountComponentAtNode()
  4. findDOMNode()
  5. createPortal()

1. **What is the purpose of render method of react-dom?**

This method is used to render a React element into the DOM in the supplied container and return a reference to the component. If the React element was previously rendered into container, it will perform an update on it and only mutate the DOM as necessary to reflect the latest changes.

ReactDOM.render(element, container[, callback])

If the optional callback is provided, it will be executed after the component is rendered or updated.

1. **What is ReactDOMServer?**

The ReactDOMServer object enables you to render components to static markup (typically used on node server). This object is mainly used for *server-side rendering* (SSR). The following methods can be used in both the server and browser environments:

* 1. renderToString()
  2. renderToStaticMarkup()

For example, you generally run a Node-based web server like Express, Hapi, or Koa, and you call renderToString to render your root component to a string, which you then send as response.

// using Express

import { renderToString } from 'react-dom/server'

import MyPage from './MyPage'

app.get('/', (req, res) => {

res.write('<!DOCTYPE html><html><head><title>My Page</title></head><body>')

res.write('<div id="content">')

res.write(renderToString(<MyPage/>))

res.write('</div></body></html>')

res.end()

})

1. **How to use innerHTML in React?**

The dangerouslySetInnerHTML attribute is React's replacement for using innerHTML in the browser DOM. Just like innerHTML, it is risky to use this attribute considering cross-site scripting (XSS) attacks. You just need to pass a \_\_html object as key and HTML text as value.

In this example MyComponent uses dangerouslySetInnerHTML attribute for setting HTML markup:

function createMarkup() {

return { \_\_html: 'First &middot; Second' }

}

function MyComponent() {

return <div dangerouslySetInnerHTML={createMarkup()} />

}

1. **How to use styles in React?**

The style attribute accepts a JavaScript object with camelCased properties rather than a CSS string. This is consistent with the DOM style JavaScript property, is more efficient, and prevents XSS security holes.

const divStyle = {

color: 'blue',

backgroundImage: 'url(' + imgUrl + ')'

};

function HelloWorldComponent() {

return <div style={divStyle}>Hello World!</div>

}

Style keys are camelCased in order to be consistent with accessing the properties on DOM nodes in JavaScript (e.g. node.style.backgroundImage).

1. **How events are different in React?**

Handling events in React elements has some syntactic differences:

* 1. React event handlers are named using camelCase, rather than lowercase.
  2. With JSX you pass a function as the event handler, rather than a string.

1. **What will happen if you use setState() in constructor?**

When you use setState(), then apart from assigning to the object state React also re-renders the component and all its children. You would get error like this: *Can only update a mounted or mounting component.* So we need to use this.state to initialize variables inside constructor.

1. **What is the impact of indexes as keys?**

Keys should be stable, predictable, and unique so that React can keep track of elements.

In the below code snippet each element's key will be based on ordering, rather than tied to the data that is being represented. This limits the optimizations that React can do.

{todos.map((todo, index) =>

<Todo

{...todo}

key={index}

/>

)}

If you use element data for unique key, assuming todo.id is unique to this list and stable, React would be able to reorder elements without needing to reevaluate them as much.

{todos.map((todo) =>

<Todo {...todo}

key={todo.id} />

)}

1. **Is it good to use setState() in componentWillMount() method?**

Yes, it is safe to use setState() inside componentWillMount() method. But at the same it is recommended to avoid async initialization in componentWillMount() lifecycle method. componentWillMount() is invoked immediately before mounting occurs. It is called before render(), therefore setting state in this method will not trigger a re-render. Avoid introducing any side-effects or subscriptions in this method. We need to make sure async calls for component initialization happened in componentDidMount() instead of componentWillMount().

componentDidMount() {

axios.get(`api/todos`)

.then((result) => {

this.setState({

messages: [...result.data]

})

})

}

1. **What will happen if you use props in initial state?**

If the props on the component are changed without the component being refreshed, the new prop value will never be displayed because the constructor function will never update the current state of the component. The initialization of state from props only runs when the component is first created.

The below component won't display the updated input value:

class MyComponent extends React.Component {

constructor(props) {

super(props)

this.state = {

records: [],

inputValue: this.props.inputValue

};

}

render() {

return <div>{this.state.inputValue}</div>

}

}

Using props inside render method will update the value:

class MyComponent extends React.Component {

constructor(props) {

super(props)

this.state = {

record: []

}

}

render() {

return <div>{this.props.inputValue}</div>

}

}

1. **How do you conditionally render components?**

In some cases you want to render different components depending on some state. JSX does not render false or undefined, so you can use conditional *short-circuiting* to render a given part of your component only if a certain condition is true.

const MyComponent = ({ name, address }) => (

<div>

<h2>{name}</h2>

{address &&

<p>{address}</p>

}

</div>

)

If you need an if-else condition then use *ternary operator*.

const MyComponent = ({ name, address }) => (

<div>

<h2>{name}</h2>

{address

? <p>{address}</p>

: <p>{'Address is not available'}</p>

}

</div>

)

1. **Why we need to be careful when spreading props on DOM elements?**

When we *spread props* we run into the risk of adding unknown HTML attributes, which is a bad practice. Instead we can use prop destructuring with ...rest operator, so it will add only required props.

For example,

const ComponentA = () =>

<ComponentB isDisplay={true} className={'componentStyle'} />

const ComponentB = ({ isDisplay, ...domProps }) =>

<div {...domProps}>{'ComponentB'}</div>

1. **How you use decorators in React?**

You can *decorate* your *class* components, which is the same as passing the component into a function. **Decorators** are flexible and readable way of modifying component functionality.

@setTitle('Profile')

class Profile extends React.Component {

//....

}

/\*

title is a string that will be set as a document title

WrappedComponent is what our decorator will receive when

put directly above a component class as seen in the example above

\*/

const setTitle = (title) => (WrappedComponent) => {

return class extends React.Component {

componentDidMount() {

document.title = title

}

render() {

return <WrappedComponent {...this.props} />

}

}

}

**Note:** Decorators are a feature that didn't make it into ES7, but are currently a *stage 2 proposal*.

1. **How do you memoize a component?**

There are memoize libraries available which can be used on function components.

For example moize library can memoize the component in another component.

import moize from 'moize'

import Component from './components/Component' // this module exports a non-memoized component

const MemoizedFoo = moize.react(Component)

const Consumer = () => {

<div>

{'I will memoize the following entry:'}

<MemoizedFoo/>

</div>

}

**Update:** Since React v16.6.0, we have a React.memo. It provides a higher order component which memoizes component unless the props change. To use it, simply wrap the component using React.memo before you use it.

const MemoComponent = React.memo(function MemoComponent(props) {

/\* render using props \*/

});

OR

export default React.memo(MyFunctionComponent);

1. **How you implement Server Side Rendering or SSR?**

React is already equipped to handle rendering on Node servers. A special version of the DOM renderer is available, which follows the same pattern as on the client side.

import ReactDOMServer from 'react-dom/server'

import App from './App'

ReactDOMServer.renderToString(<App />)

This method will output the regular HTML as a string, which can be then placed inside a page body as part of the server response. On the client side, React detects the pre-rendered content and seamlessly picks up where it left off.

1. **How to enable production mode in React?**

You should use Webpack's DefinePlugin method to set NODE\_ENV to production, by which it strip out things like propType validation and extra warnings. Apart from this, if you minify the code, for example, Uglify's dead-code elimination to strip out development only code and comments, it will drastically reduce the size of your bundle.

1. **What is CRA and its benefits?**

The create-react-app CLI tool allows you to quickly create & run React applications with no configuration step.

Let's create Todo App using *CRA*:

# Installation

$ npm install -g create-react-app

# Create new project

$ create-react-app todo-app

$ cd todo-app

# Build, test and run

$ npm run build

$ npm run test

$ npm start

It includes everything we need to build a React app:

* 1. React, JSX, ES6, and Flow syntax support.
  2. Language extras beyond ES6 like the object spread operator.
  3. Autoprefixed CSS, so you don’t need -webkit- or other prefixes.
  4. A fast interactive unit test runner with built-in support for coverage reporting.
  5. A live development server that warns about common mistakes.
  6. A build script to bundle JS, CSS, and images for production, with hashes and sourcemaps.

1. **What is the lifecycle methods order in mounting?**

The lifecycle methods are called in the following order when an instance of a component is being created and inserted into the DOM.

* 1. constructor()
  2. static getDerivedStateFromProps()
  3. render()
  4. componentDidMount()

1. **What are the lifecycle methods going to be deprecated in React v16?**

The following lifecycle methods going to be unsafe coding practices and will be more problematic with async rendering.

* 1. componentWillMount()
  2. componentWillReceiveProps()
  3. componentWillUpdate()

Starting with React v16.3 these methods are aliased with UNSAFE\_ prefix, and the unprefixed version will be removed in React v17.

1. **What is the purpose of getDerivedStateFromProps() lifecycle method?**

The new static getDerivedStateFromProps() lifecycle method is invoked after a component is instantiated as well as before it is re-rendered. It can return an object to update state, or null to indicate that the new props do not require any state updates.

class MyComponent extends React.Component {

static getDerivedStateFromProps(props, state) {

// ...

}

}

This lifecycle method along with componentDidUpdate() covers all the use cases of componentWillReceiveProps().

1. **What is the purpose of getSnapshotBeforeUpdate() lifecycle method?**

The new getSnapshotBeforeUpdate() lifecycle method is called right before DOM updates. The return value from this method will be passed as the third parameter to componentDidUpdate().

class MyComponent extends React.Component {

getSnapshotBeforeUpdate(prevProps, prevState) {

// ...

}

}

This lifecycle method along with componentDidUpdate() covers all the use cases of componentWillUpdate().

1. **Do Hooks replace render props and higher order components?**

Both render props and higher-order components render only a single child but in most of the cases Hooks are a simpler way to serve this by reducing nesting in your tree.

1. **What is the recommended way for naming components?**

It is recommended to name the component by reference instead of using displayName.

Using displayName for naming component:

export default React.createClass({

displayName: 'TodoApp',

// ...

})

The **recommended** approach:

export default class TodoApp extends React.Component {

// ...

}

1. **What is the recommended ordering of methods in component class?**

*Recommended* ordering of methods from *mounting* to *render stage*:

* 1. static methods
  2. constructor()
  3. getChildContext()
  4. componentWillMount()
  5. componentDidMount()
  6. componentWillReceiveProps()
  7. shouldComponentUpdate()
  8. componentWillUpdate()
  9. componentDidUpdate()
  10. componentWillUnmount()
  11. click handlers or event handlers like onClickSubmit() or onChangeDescription()
  12. getter methods for render like getSelectReason() or getFooterContent()
  13. optional render methods like renderNavigation() or renderProfilePicture()
  14. render()

1. **What is a switching component?**

A *switching component* is a component that renders one of many components. We need to use object to map prop values to components.

For example, a switching component to display different pages based on page prop:

import HomePage from './HomePage'

import AboutPage from './AboutPage'

import ServicesPage from './ServicesPage'

import ContactPage from './ContactPage'

const PAGES = {

home: HomePage,

about: AboutPage,

services: ServicesPage,

contact: ContactPage

}

const Page = (props) => {

const Handler = PAGES[props.page] || ContactPage

return <Handler {...props} />

}

// The keys of the PAGES object can be used in the prop types to catch dev-time errors.

Page.propTypes = {

page: PropTypes.oneOf(Object.keys(PAGES)).isRequired

}

1. **Why we need to pass a function to setState()?**

The reason behind for this is that setState() is an asynchronous operation. React batches state changes for performance reasons, so the state may not change immediately after setState() is called. That means you should not rely on the current state when calling setState() since you can't be sure what that state will be. The solution is to pass a function to setState(), with the previous state as an argument. By doing this you can avoid issues with the user getting the old state value on access due to the asynchronous nature of setState().

Let's say the initial count value is zero. After three consecutive increment operations, the value is going to be incremented only by one.

// assuming this.state.count === 0

this.setState({ count: this.state.count + 1 })

this.setState({ count: this.state.count + 1 })

this.setState({ count: this.state.count + 1 })

// this.state.count === 0, not 3

If we pass a function to setState(), the count gets incremented correctly.

this.setState((prevState, props) => ({

count: prevState.count + props.increment

}))

// this.state.count === 3 as expected

**(OR)**

**Why function is preferred over object for setState()?**

React may batch multiple setState() calls into a single update for performance. Because this.props and this.state may be updated asynchronously, you should not rely on their values for calculating the next state.

This counter example will fail to update as expected:

// Wrong

this.setState({

counter: this.state.counter + this.props.increment,

})

The preferred approach is to call setState() with function rather than 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((prevState, props) => ({

counter: prevState.counter + props.increment

}))

1. **What is strict mode in React?**

React.StrictMode is a useful component for highlighting potential problems in an application. Just like <Fragment>, <StrictMode> does not render any extra DOM elements. It activates additional checks and warnings for its descendants. These checks apply for *development mode* only.

import React from 'react'

function ExampleApplication() {

return (

<div>

<Header />

<React.StrictMode>

<div>

<ComponentOne />

<ComponentTwo />

</div>

</React.StrictMode>

<Header />

</div>

)

}

In the example above, the *strict mode* checks apply to <ComponentOne> and <ComponentTwo> components only.

1. **What are React Mixins?**

*Mixins* are a way to totally separate components to have a common functionality. Mixins **should not be used** and can be replaced with *higher-order components* or *decorators*.

One of the most commonly used mixins is PureRenderMixin. You might be using it in some components to prevent unnecessary re-renders when the props and state are shallowly equal to the previous props and state:

const PureRenderMixin = require('react-addons-pure-render-mixin')

const Button = React.createClass({

mixins: [PureRenderMixin],

// ...

})

1. **Why is isMounted() an anti-pattern and what is the proper solution?**

The primary use case for isMounted() is to avoid calling setState() after a component has been unmounted, because it will emit a warning.

if (this.isMounted()) {

this.setState({...})

}

Checking isMounted() before calling setState() does eliminate the warning, but it also defeats the purpose of the warning. Using isMounted() is a code smell because the only reason you would check is because you think you might be holding a reference after the component has unmounted.

An optimal solution would be to find places where setState() might be called after a component has unmounted, and fix them. Such situations most commonly occur due to callbacks, when a component is waiting for some data and gets unmounted before the data arrives. Ideally, any callbacks should be canceled in componentWillUnmount(), prior to unmounting.

1. **What are the Pointer Events supported in React?**

*Pointer Events* provide a unified way of handling all input events. In the old days we had a mouse and respective event listeners to handle them but nowadays we have many devices which don't correlate to having a mouse, like phones with touch surface or pens. We need to remember that these events will only work in browsers that support the *Pointer Events* specification.

The following event types are now available in *React DOM*:

* 1. onPointerDown
  2. onPointerMove
  3. onPointerUp
  4. onPointerCancel
  5. onGotPointerCapture
  6. onLostPointerCapture
  7. onPointerEnter
  8. onPointerLeave
  9. onPointerOver
  10. onPointerOut

1. **Why should component names start with capital letter?**

If you are rendering your component using JSX, the name of that component has to begin with a capital letter otherwise React will throw an error as unrecognized tag. This convention is because only HTML elements and SVG tags can begin with a lowercase letter.

class SomeComponent extends Component {

// Code goes here

}

You can define component class which name starts with lowercase letter, but when it's imported it should have capital letter. Here lowercase is fine:

class myComponent extends Component {

render() {

return <div />

}

}

export default myComponent

While when imported in another file it should start with capital letter:

import MyComponent from './MyComponent'

**What are the exceptions on React component naming?**

The component names should start with a uppercase letter but there are few exceptions on this convention. The lowercase tag names with a dot (property accessors) are still considered as valid component names. For example the below tag can be compiled to a valid component,

render() {

return (

<obj.component/> // `React.createElement(obj.component)`

)

}

1. **Are custom DOM attributes supported in React v16?**

Yes. In the past, React used to ignore unknown DOM attributes. If you wrote JSX with an attribute that React doesn't recognize, React would just skip it.

For example, let's take a look at the below attribute:

<div mycustomattribute={'something'} />

Would render an empty div to the DOM with React v15:

<div />

In React v16 any unknown attributes will end up in the DOM:

<div mycustomattribute='something' />

This is useful for supplying browser-specific non-standard attributes, trying new DOM APIs, and integrating with opinionated third-party libraries.

1. **What is the difference between constructor and getInitialState?**

You should initialize state in the constructor when using ES6 classes, and getInitialState() method when using React.createClass().

**Using ES6 classes:**

class MyComponent extends React.Component {

constructor(props) {

super(props)

this.state = { /\* initial state \*/ }

}

}

**Using React.createClass():**

const MyComponent = React.createClass({

getInitialState() {

return { /\* initial state \*/ }

}

})

**Note:** React.createClass() is deprecated and removed in React v16. Use plain JavaScript classes instead.

1. **Can you force a component to re-render without calling setState?**

By default, when your component's state or props change, your component will re-render. If your render() method depends on some other data, you can tell React that the component needs re-rendering by calling forceUpdate().

component.forceUpdate(callback)

It is recommended to avoid all uses of forceUpdate() and only read from this.props and this.state in render().

1. **What is the difference between super() and super(props) in React using ES6 classes?**

When you want to access this.props in constructor() then you should pass props to super() method.

**Using super(props):**

class MyComponent extends React.Component {

constructor(props) {

super(props)

console.log(this.props) // { name: 'John', ... }

}

}

**Using super():**

class MyComponent extends React.Component {

constructor(props) {

super()

console.log(this.props) // undefined

}

}

Outside constructor() both will display same value for this.props.

1. **How to loop inside JSX?**

You can simply use Array.prototype.map with ES6 *arrow function* syntax.

For example, the items array of objects is mapped into an array of components:

<tbody>

{items.map(item => <SomeComponent key={item.id} name={item.name} />)}

</tbody>

But you can't iterate using for loop:

<tbody>

for (let i = 0; i < items.length; i++) {

<SomeComponent key={items[i].id} name={items[i].name} />

}

</tbody>

This is because JSX tags are transpiled into *function calls*, and you can't use statements inside expressions. This may change thanks to do expressions which are *stage 1 proposal*.

1. **How do you access props in attribute quotes?**

React (or JSX) doesn't support variable interpolation inside an attribute value. The below representation won't work:

<img className='image' src='images/{this.props.image}' />

But you can put any JS expression inside curly braces as the entire attribute value. So the below expression works:

<img className='image' src={'images/' + this.props.image} />

Using *template strings* will also work:

<img className='image' src={`images/${this.props.image}`} />

1. **What is React proptype array with shape?**

If you want to pass an array of objects to a component with a particular shape then use React.PropTypes.shape() as an argument to React.PropTypes.arrayOf().

ReactComponent.propTypes = {

arrayWithShape: React.PropTypes.arrayOf(React.PropTypes.shape({

color: React.PropTypes.string.isRequired,

fontSize: React.PropTypes.number.isRequired

})).isRequired

}

1. **How to conditionally apply class attributes?**

You shouldn't use curly braces inside quotes because it is going to be evaluated as a string.

<div className="btn-panel {this.props.visible ? 'show' : 'hidden'}">

Instead you need to move curly braces outside (don't forget to include spaces between class names):

<div className={'btn-panel ' + (this.props.visible ? 'show' : 'hidden')}>

*Template strings* will also work:

<div className={`btn-panel ${this.props.visible ? 'show' : 'hidden'}`}>

1. **What is the difference between React and ReactDOM?**

The react package contains React.createElement(), React.Component, React.Children, and other helpers related to elements and component classes. You can think of these as the isomorphic or universal helpers that you need to build components. The react-dom package contains ReactDOM.render(), and in react-dom/server we have *server-side rendering* support with ReactDOMServer.renderToString() and ReactDOMServer.renderToStaticMarkup().

1. **Why ReactDOM is separated from React?**

The React team worked on extracting all DOM-related features into a separate library called *ReactDOM*. React v0.14 is the first release in which the libraries are split. By looking at some of the packages, react-native, react-art, react-canvas, and react-three, it has become clear that the beauty and essence of React has nothing to do with browsers or the DOM.

To build more environments that React can render to, React team planned to split the main React package into two: react and react-dom. This paves the way to writing components that can be shared between the web version of React and React Native.

1. **How to use React label element?**

If you try to render a <label> element bound to a text input using the standard for attribute, then it produces HTML missing that attribute and prints a warning to the console.

<label for={'user'}>{'User'}</label>

<input type={'text'} id={'user'} />

Since for is a reserved keyword in JavaScript, use htmlFor instead.

<label htmlFor={'user'}>{'User'}</label>

<input type={'text'} id={'user'} />

1. **How to combine multiple inline style objects?**

You can use *spread operator* in regular React:

<button style={{...styles.panel.button, ...styles.panel.submitButton}}>{'Submit'}</button>

If you're using React Native then you can use the array notation:

<button style={[styles.panel.button, styles.panel.submitButton]}>{'Submit'}</button>

1. **How to re-render the view when the browser is resized?**

You can listen to the resize event in componentDidMount() and then update the dimensions (width and height). You should remove the listener in componentWillUnmount() method.

class WindowDimensions extends React.Component {

constructor(props){

super(props);

this.updateDimensions = this.updateDimensions.bind(this);

}

componentWillMount() {

this.updateDimensions()

}

componentDidMount() {

window.addEventListener('resize', this.updateDimensions)

}

componentWillUnmount() {

window.removeEventListener('resize', this.updateDimensions)

}

updateDimensions() {

this.setState({width: window.innerWidth, height: window.innerHeight})

}

render() {

return <span>{this.state.width} x {this.state.height}</span>

}

}

1. **What is the difference between setState() and replaceState() methods?**

When you use setState() the current and previous states are merged. replaceState() throws out the current state, and replaces it with only what you provide. Usually setState() is used unless you really need to remove all previous keys for some reason. You can also set state to false/null in setState() instead of using replaceState().

1. **How to listen to state changes?**

The componentDidUpdate lifecycle method will be called when state changes. You can compare provided state and props values with current state and props to determine if something meaningful changed.

componentDidUpdate(object prevProps, object prevState)

**Note:** The previous releases of ReactJS also uses componentWillUpdate(object nextProps, object nextState) for state changes. It has been deprecated in latest releases.

1. **What is the recommended approach of removing an array element in React state?**

The better approach is to use Array.prototype.filter() method.

For example, let's create a removeItem() method for updating the state.

removeItem(index) {

this.setState({

data: this.state.data.filter((item, i) => i !== index)

})

}

1. **Is it possible to use React without rendering HTML?**

It is possible with latest version (>=16.2). Below are the possible options:

render() {

return false

}

render() {

return null

}

render() {

return []

}

render() {

return <React.Fragment></React.Fragment>

}

render() {

return <></>

}

Returning undefined won't work.

1. **How to pretty print JSON with React?**

We can use <pre> tag so that the formatting of the JSON.stringify() is retained:

const data = { name: 'John', age: 42 }

class User extends React.Component {

render() {

return (

<pre>

{JSON.stringify(data, null, 2)}

</pre>

)

}

}

React.render(<User />, document.getElementById('container'))

1. **Why you can't update props in React?**

The React philosophy is that props should be *immutable* and *top-down*. This means that a parent can send any prop values to a child, but the child can't modify received props.

1. **How to focus an input element on page load?**

You can do it by creating *ref* for input element and using it in componentDidMount():

class App extends React.Component{

componentDidMount() {

this.nameInput.focus()

}

render() {

return (

<div>

<input

defaultValue={'Won\'t focus'}

/>

<input

ref={(input) => this.nameInput = input}

defaultValue={'Will focus'}

/>

</div>

)

}

}

ReactDOM.render(<App />, document.getElementById('app'))

1. **What are the possible ways of updating objects in state?**
   1. **Calling setState() with an object to merge with state:**
      * Using Object.assign() to create a copy of the object:
      * const user = Object.assign({}, this.state.user, { age: 42 })

this.setState({ user })

* + - Using *spread operator*:
    - const user = { ...this.state.user, age: 42 }

this.setState({ user })

* 1. **Calling setState() with a function:**
  2. this.setState(prevState => ({
  3. user: {
  4. ...prevState.user,
  5. age: 42
  6. }

}))

1. **How can we find the version of React at runtime in the browser?**

You can use React.version to get the version.

const REACT\_VERSION = React.version

ReactDOM.render(

<div>{`React version: ${REACT\_VERSION}`}</div>,

document.getElementById('app')

)

1. **What are the approaches to include polyfills in your create-react-app?**

There are approaches to include polyfills in create-react-app,

* 1. **Manual import from core-js:**

Create a file called (something like) polyfills.js and import it into root index.js file. Run npm install core-js or yarn add core-js and import your specific required features.

import 'core-js/fn/array/find'

import 'core-js/fn/array/includes'

import 'core-js/fn/number/is-nan'

* 1. **Using Polyfill service:**

Use the polyfill.io CDN to retrieve custom, browser-specific polyfills by adding this line to index.html:

<script src='https://cdn.polyfill.io/v2/polyfill.min.js?features=default,Array.prototype.includes'></script>

In the above script we had to explicitly request the Array.prototype.includes feature as it is not included in the default feature set.

1. **How to use https instead of http in create-react-app?**

You just need to use HTTPS=true configuration. You can edit your package.json scripts section:

"scripts": {

"start": "set HTTPS=true && react-scripts start"

}

or just run set HTTPS=true && npm start

1. **How to avoid using relative path imports in create-react-app?**

Create a file called .env in the project root and write the import path:

NODE\_PATH=src/app

After that restart the development server. Now you should be able to import anything inside src/app without relative paths.

1. **How to add Google Analytics for React Router?**

Add a listener on the history object to record each page view:

history.listen(function (location) {

window.ga('set', 'page', location.pathname + location.search)

window.ga('send', 'pageview', location.pathname + location.search)

})

1. **How to update a component every second?**

You need to use setInterval() to trigger the change, but you also need to clear the timer when the component unmounts to prevent errors and memory leaks.

componentDidMount() {

this.interval = setInterval(() => this.setState({ time: Date.now() }), 1000)

}

componentWillUnmount() {

clearInterval(this.interval)

}

1. **How do you apply vendor prefixes to inline styles in React?**

React *does not* apply *vendor prefixes* automatically. You need to add vendor prefixes manually.

<div style={{

transform: 'rotate(90deg)',

WebkitTransform: 'rotate(90deg)', // note the capital 'W' here

msTransform: 'rotate(90deg)' // 'ms' is the only lowercase vendor prefix

}} />

1. **How to import and export components using React and ES6?**

You should use default for exporting the components

import React from 'react'

import User from 'user'

export default class MyProfile extends React.Component {

render(){

return (

<User type="customer">

//...

</User>

)

}

}

With the export specifier, the MyProfile is going to be the member and exported to this module and the same can be imported without mentioning the name in other components.

1. **Why is a component constructor called only once?**

React's *reconciliation* algorithm assumes that without any information to the contrary, if a custom component appears in the same place on subsequent renders, it's the same component as before, so reuses the previous instance rather than creating a new one.

1. **How to define constants in React?**

You can use ES7 static field to define constant.

class MyComponent extends React.Component {

static DEFAULT\_PAGINATION = 10

}

*Static fields* are part of the *Class Fields* stage 3 proposal.

1. **How to programmatically trigger click event in React?**

You could use the ref prop to acquire a reference to the underlying HTMLInputElement object through a callback, store the reference as a class property, then use that reference to later trigger a click from your event handlers using the HTMLElement.click method.

This can be done in two steps:

* 1. Create ref in render method:

<input ref={input => this.inputElement = input} />

* 1. Apply click event in your event handler:

this.inputElement.click()

1. **Is it possible to use async/await in plain React?**

If you want to use async/await in React, you will need *Babel* and transform-async-to-generator plugin. React Native ships with Babel and a set of transforms.

1. **What are the common folder structures for React?**

There are two common practices for React project file structure.

* 1. **Grouping by features or routes:**

One common way to structure projects is locate CSS, JS, and tests together, grouped by feature or route.

common/

├─ Avatar.js

├─ Avatar.css

├─ APIUtils.js

└─ APIUtils.test.js

feed/

├─ index.js

├─ Feed.js

├─ Feed.css

├─ FeedStory.js

├─ FeedStory.test.js

└─ FeedAPI.js

profile/

├─ index.js

├─ Profile.js

├─ ProfileHeader.js

├─ ProfileHeader.css

└─ ProfileAPI.js

* 1. **Grouping by file type:**

Another popular way to structure projects is to group similar files together.

api/

├─ APIUtils.js

├─ APIUtils.test.js

├─ ProfileAPI.js

└─ UserAPI.js

components/

├─ Avatar.js

├─ Avatar.css

├─ Feed.js

├─ Feed.css

├─ FeedStory.js

├─ FeedStory.test.js

├─ Profile.js

├─ ProfileHeader.js

└─ ProfileHeader.css

1. **What are the popular packages for animation?**

*React Transition Group* and *React Motion* are popular animation packages in React ecosystem.

1. **What is the benefit of styles modules?**

It is recommended to avoid hard coding style values in components. Any values that are likely to be used across different UI components should be extracted into their own modules.

For example, these styles could be extracted into a separate component:

export const colors = {

white,

black,

blue

}

export const space = [

0,

8,

16,

32,

64

]

And then imported individually in other components:

import { space, colors } from './styles'

1. **What are the popular React-specific linters?**

ESLint is a popular JavaScript linter. There are plugins available that analyse specific code styles. One of the most common for React is an npm package called eslint-plugin-react. By default, it will check a number of best practices, with rules checking things from keys in iterators to a complete set of prop types.

Another popular plugin is eslint-plugin-jsx-a11y, which will help fix common issues with accessibility. As JSX offers slightly different syntax to regular HTML, issues with alt text and tabindex, for example, will not be picked up by regular plugins.

1. **How to make AJAX call and in which component lifecycle methods should I make an AJAX call?**

You can use AJAX libraries such as Axios, jQuery AJAX, and the browser built-in fetch. You should fetch data in the componentDidMount() lifecycle method. This is so you can use setState() to update your component when the data is retrieved.

For example, the employees list fetched from API and set local state:

class MyComponent extends React.Component {

constructor(props) {

super(props)

this.state = {

employees: [],

error: null

}

}

componentDidMount() {

fetch('https://api.example.com/items')

.then(res => res.json())

.then(

(result) => {

this.setState({

employees: result.employees

})

},

(error) => {

this.setState({ error })

}

)

}

render() {

const { error, employees } = this.state

if (error) {

return <div>Error: {error.message}</div>;

} else {

return (

<ul>

{employees.map(employee => (

<li key={employee.name}>

{employee.name}-{employee.experience}

</li>

))}

</ul>

)

}

}

}

1. **What are render props?**

**Render Props** is a simple technique for sharing code between components using a prop whose value is a function. The below component uses render prop which returns a React element.

<DataProvider render={data => (

<h1>{`Hello ${data.target}`}</h1>

)}/>

Libraries such as React Router and DownShift are using this pattern.

**React Router**

1. **What is React Router?**

React Router is a powerful routing library built on top of React that helps you add new screens and flows to your application incredibly quickly, all while keeping the URL in sync with what's being displayed on the page.

1. **How React Router is different from history library?**

React Router is a wrapper around the history library which handles interaction with the browser's window.history with its browser and hash histories. It also provides memory history which is useful for environments that don't have global history, such as mobile app development (React Native) and unit testing with Node.

1. **What are the <Router> components of React Router v4?**

React Router v4 provides below 3 <Router> components:

* 1. <BrowserRouter>
  2. <HashRouter>
  3. <MemoryRouter>

The above components will create *browser*, *hash*, and *memory* history instances. React Router v4 makes the properties and methods of the history instance associated with your router available through the context in the router object.

1. **What is the purpose of push() and replace() methods of history?**

A history instance has two methods for navigation purpose.

* 1. push()
  2. replace()

If you think of the history as an array of visited locations, push() will add a new location to the array and replace() will replace the current location in the array with the new one.

1. **How do you programmatically navigate using React Router v4?**

There are three different ways to achieve programmatic routing/navigation within components.

* 1. **Using the withRouter() higher-order function:**

The withRouter() higher-order function will inject the history object as a prop of the component. This object provides push() and replace() methods to avoid the usage of context.

import { withRouter } from 'react-router-dom' // this also works with 'react-router-native'

const Button = withRouter(({ history }) => (

<button

type='button'

onClick={() => { history.push('/new-location') }}

>

{'Click Me!'}

</button>

))

* 1. **Using <Route> component and render props pattern:**

The <Route> component passes the same props as withRouter(), so you will be able to access the history methods through the history prop.

import { Route } from 'react-router-dom'

const Button = () => (

<Route render={({ history }) => (

<button

type='button'

onClick={() => { history.push('/new-location') }}

>

{'Click Me!'}

</button>

)} />

)

* 1. **Using context:**

This option is not recommended and treated as unstable API.

const Button = (props, context) => (

<button

type='button'

onClick={() => {

context.history.push('/new-location')

}}

>

{'Click Me!'}

</button>

)

Button.contextTypes = {

history: React.PropTypes.shape({

push: React.PropTypes.func.isRequired

})

}

1. **How to get query parameters in React Router v4?**

The ability to parse query strings was taken out of React Router v4 because there have been user requests over the years to support different implementation. So the decision has been given to users to choose the implementation they like. The recommended approach is to use query strings library.

const queryString = require('query-string');

const parsed = queryString.parse(props.location.search);

You can also use URLSearchParams if you want something native:

const params = new URLSearchParams(props.location.search)

const foo = params.get('name')

You should use a *polyfill* for IE11.

1. **Why you get "Router may have only one child element" warning?**

You have to wrap your Route's in a <Switch> block because <Switch> is unique in that it renders a route exclusively.

At first you need to add Switch to your imports:

import { Switch, Router, Route } from 'react-router'

Then define the routes within <Switch> block:

<Router>

<Switch>

<Route {/\* ... \*/} />

<Route {/\* ... \*/} />

</Switch>

</Router>

1. **How to pass params to history.push method in React Router v4?**

While navigating you can pass props to the history object:

this.props.history.push({

pathname: '/template',

search: '?name=sudheer',

state: { detail: response.data }

})

The search property is used to pass query params in push() method.

1. **How to implement *default* or *NotFound* page?**

A <Switch> renders the first child <Route> that matches. A <Route> with no path always matches. So you just need to simply drop path attribute as below

<Switch>

<Route exact path="/" component={Home}/>

<Route path="/user" component={User}/>

<Route component={NotFound} />

</Switch>

1. **How to get history on React Router v4?**

Below are the list of steps to get history object on React Router v4,

* 1. Create a module that exports a history object and import this module across the project.

For example, create history.js file:

import { createBrowserHistory } from 'history'

export default createBrowserHistory({

/\* pass a configuration object here if needed \*/

})

* 1. You should use the <Router> component instead of built-in routers. Imported the above history.js inside index.js file:
  2. import { Router } from 'react-router-dom'
  3. import history from './history'
  4. import App from './App'
  5. ReactDOM.render((
  6. <Router history={history}>
  7. <App />
  8. </Router>

), holder)

* 1. You can also use push method of history object similar to built-in history object:
  2. // some-other-file.js
  3. import history from './history'

history.push('/go-here')

1. **How to perform automatic redirect after login?**

The react-router package provides <Redirect> component in React Router. Rendering a <Redirect> will navigate to a new location. Like server-side redirects, the new location will override the current location in the history stack.

import React, { Component } from 'react'

import { Redirect } from 'react-router'

export default class LoginComponent extends Component {

render() {

if (this.state.isLoggedIn === true) {

return <Redirect to="/your/redirect/page" />

} else {

return <div>{'Login Please'}</div>

}

}

}

**React Internationalization**

1. **What is React Intl?**

The *React Intl* library makes internalization in React straightforward, with off-the-shelf components and an API that can handle everything from formatting strings, dates, and numbers, to pluralization. React Intl is part of *FormatJS* which provides bindings to React via its components and API.

1. **What are the main features of React Intl?**

Below are the main features of React Intl,

* 1. Display numbers with separators.
  2. Display dates and times correctly.
  3. Display dates relative to "now".
  4. Pluralize labels in strings.
  5. Support for 150+ languages.
  6. Runs in the browser and Node.
  7. Built on standards.

1. **What are the two ways of formatting in React Intl?**

The library provides two ways to format strings, numbers, and dates:

* 1. **Using react components:**
  2. <FormattedMessage
  3. id={'account'}
  4. defaultMessage={'The amount is less than minimum balance.'}

/>

* 1. **Using an API:**
  2. const messages = defineMessages({
  3. accountMessage: {
  4. id: 'account',
  5. defaultMessage: 'The amount is less than minimum balance.',
  6. }
  7. })

formatMessage(messages.accountMessage)

1. **How to use <FormattedMessage> as placeholder using React Intl?**

The <Formatted... /> components from react-intl return elements, not plain text, so they can't be used for placeholders, alt text, etc. In that case, you should use lower level API formatMessage(). You can inject the intl object into your component using injectIntl() higher-order component and then format the message using formatMessage() available on that object.

import React from 'react'

import { injectIntl, intlShape } from 'react-intl'

const MyComponent = ({ intl }) => {

const placeholder = intl.formatMessage({id: 'messageId'})

return <input placeholder={placeholder} />

}

MyComponent.propTypes = {

intl: intlShape.isRequired

}

export default injectIntl(MyComponent)

1. **How to access current locale with React Intl?**

You can get the current locale in any component of your application using injectIntl():

import { injectIntl, intlShape } from 'react-intl'

const MyComponent = ({ intl }) => (

<div>{`The current locale is ${intl.locale}`}</div>

)

MyComponent.propTypes = {

intl: intlShape.isRequired

}

export default injectIntl(MyComponent)

1. **How to format date using React Intl?**

The injectIntl() higher-order component will give you access to the formatDate() method via the props in your component. The method is used internally by instances of FormattedDate and it returns the string representation of the formatted date.

import { injectIntl, intlShape } from 'react-intl'

const stringDate = this.props.intl.formatDate(date, {

year: 'numeric',

month: 'numeric',

day: 'numeric'

})

const MyComponent = ({intl}) => (

<div>{`The formatted date is ${stringDate}`}</div>

)

MyComponent.propTypes = {

intl: intlShape.isRequired

}

export default injectIntl(MyComponent)

**React Testing**

1. **What is Shallow Renderer in React testing?**

*Shallow rendering* is useful for writing unit test cases in React. It lets you render a component *one level deep* and assert facts about what its render method returns, without worrying about the behavior of child components, which are not instantiated or rendered.

For example, if you have the following component:

function MyComponent() {

return (

<div>

<span className={'heading'}>{'Title'}</span>

<span className={'description'}>{'Description'}</span>

</div>

)

}

Then you can assert as follows:

import ShallowRenderer from 'react-test-renderer/shallow'

// in your test

const renderer = new ShallowRenderer()

renderer.render(<MyComponent />)

const result = renderer.getRenderOutput()

expect(result.type).toBe('div')

expect(result.props.children).toEqual([

<span className={'heading'}>{'Title'}</span>,

<span className={'description'}>{'Description'}</span>

])

1. **What is TestRenderer package in React?**

This package provides a renderer that can be used to render components to pure JavaScript objects, without depending on the DOM or a native mobile environment. This package makes it easy to grab a snapshot of the platform view hierarchy (similar to a DOM tree) rendered by a ReactDOM or React Native without using a browser or jsdom.

import TestRenderer from 'react-test-renderer'

const Link = ({page, children}) => <a href={page}>{children}</a>

const testRenderer = TestRenderer.create(

<Link page={'https://www.facebook.com/'}>{'Facebook'}</Link>

)

console.log(testRenderer.toJSON())

// {

// type: 'a',

// props: { href: 'https://www.facebook.com/' },

// children: [ 'Facebook' ]

// }

1. **What is the purpose of ReactTestUtils package?**

*ReactTestUtils* are provided in the with-addons package and allow you to perform actions against a simulated DOM for the purpose of unit testing.

1. **What is Jest?**

*Jest* is a JavaScript unit testing framework created by Facebook based on Jasmine and provides automated mock creation and a jsdom environment. It's often used for testing components.

1. **What are the advantages of Jest over Jasmine?**

There are couple of advantages compared to Jasmine:

* + Automatically finds tests to execute in your source code.
  + Automatically mocks dependencies when running your tests.
  + Allows you to test asynchronous code synchronously.
  + Runs your tests with a fake DOM implementation (via jsdom) so that your tests can be run on the command line.
  + Runs tests in parallel processes so that they finish sooner.

1. **Give a simple example of Jest test case**

Let's write a test for a function that adds two numbers in sum.js file:

const sum = (a, b) => a + b

export default sum

Create a file named sum.test.js which contains actual test:

import sum from './sum'

test('adds 1 + 2 to equal 3', () => {

expect(sum(1, 2)).toBe(3)

})

And then add the following section to your package.json:

{

"scripts": {

"test": "jest"

}

}

Finally, run yarn test or npm test and Jest will print a result:

$ yarn test

PASS ./sum.test.js

✓ adds 1 + 2 to equal 3 (2ms)