<https://tylermcginnis.com/react-interview-questions/>

<https://tylermcginnis.com/react-aha-moments/>

[ttps://camjackson.net/post/9-things-every-reactjs-beginner-should-know](https://camjackson.net/post/9-things-every-reactjs-beginner-should-know#write-stateless-components)

Composition over inheritane - <https://www.youtube.com/watch?v=wfMtDGfHWpA>

**Why did React was created? Isnt a MVC framework? Does it use templates ?**

<https://reactjs.org/blog/2013/06/05/why-react.html>

**Design principles**

important to us that you can add functionality to a component without causing rippling changes throughout the codebase.

For example, it should be possible to introduce some local state into a component without changing any of the components using it.

In general we [resist adding features](https://www.youtube.com/watch?v=4anAwXYqLG8). don’t want to bloat your apps with useless library code.

For example, if React didn’t provide support for local state or lifecycle hooks, people would create custom abstractions. This is why sometimes we add features to React itself. If we notice that many components implement a certain feature in incompatible or inefficient ways, we might prefer to bake it into React. We don’t do it lightly. When we do it, it’s because we are confident that raising the abstraction level benefits the whole ecosystem. State, lifecycle hooks, cross-browser event normalization are good examples of this.

### Interoperability

Facebook has a massive non-React codebase. Its website uses a mix of a server-side component system called XHP, internal UI libraries that came before React, and React itself. It is important to us that any product team can [start using React for a small feature](https://www.youtube.com/watch?v=BF58ZJ1ZQxY) rather than rewrite their code to bet on it.

This is why React provides escape hatches to work with mutable models, and tries to work well together with other UI libraries. You can wrap an existing imperative UI into a declarative component, and vice versa. This is crucial for gradual adoption.

React does not want to be fully “reactive”. Using pull instead of Push paragidm.

### Debugging

When something goes wrong, it is important that you have breadcrumbs to trace the mistake to its source in the codebase. In React, props and state are those breadcrumbs.

If the props are wrong, you can traverse the tree up in the inspector, looking for the component that first “poisoned the well” by passing bad props down.

This doesn’t mean that we ignore the issues raised by the community. For example, we added support for [web components](https://reactjs.org/docs/webcomponents.html) and [SVG](https://github.com/facebook/react/pull/6243) to React even though we don’t rely on either of them internally. We are actively [listening to your pain points](https://github.com/facebook/react/issues/2686) and [address them](https://reactjs.org/blog/2016/07/11/introducing-reacts-error-code-system.html).

we use componentDidMount() instead of didMount() or onMount(). This is [intentional](https://github.com/reactjs/react-future/issues/40#issuecomment-142442124). The goal is to make the points of interaction with the library highly visible.

<https://reactjs.org/docs/design-principles.html>

**Wipro questions**

1. **Route render if path is matched what to call render or component**

Route to match home component

=========

1. **Jest** =>

[Enzyme](http://airbnb.io/enzyme/) is a common tool in the React ecosystem that makes it easier to write tests for how components will behave. By default, our application includes a library called jsdom to allow us to simulate the DOM and test its runtime behavior without a browser. Enzyme is similar, but builds on jsdom and makes it easier to make certain queries about our components.

<https://github.com/Microsoft/TypeScript-React-Starter#writing-tests-with-jest>

Shallow , jest.fn to spy on methods, await aysnc to test async call, expect to matchers

Shallow rendering is nice, because it allows you to render a single component completely, but without delving into any of its child components to render those. Instead, the resulting object will tell you things like the type and props of the children. This gives us good isolation, allowing testing of a single component at a time.

Stateless components are the ones easy to test.

<https://camjackson.net/post/9-things-every-reactjs-beginner-should-know#write-stateless-components> Beforeall => constructor componentwillmount,

Beforeeach => every single run before **it** function

mockimpllemention  
Jest with parallel ism diff

1. **Pure function**

amount.total =-val return amount.total

1. The function always returns the same result if the same arguments are passed in. It does not depend on any state, or data, change during a program’s execution. It must only depend on its input arguments.
2. The function does not produce any observable side effects such as network requests, input and output devices, or data mutation.

Side effects include, but are not limited to:

* Making a HTTP request
* Mutating data
* Printing to a screen or console
* DOM Query/Manipulation
* Math.random()
* Getting the current time

function priceAfterTax(productPrice) {  
 return (productPrice \* 0.20) + productPrice;  
}

It passes both 1, and 2, of the requirements for a function to be declared pure. It doesn’t depend on any external input, it doesn’t mutate any data and it doesn’t have any side effects.

var tax = 20;

function calculateTax(productPrice) {  
 return (productPrice \* (tax/100)) + productPrice;   
}

IMPURE because the function depends on an external tax variable you’d be right! A pure function cannot depend on outside variables.

4. Check pure function for object = > see anjana video

5. Input null or undefined initially then populate value later ==> <https://stackblitz.com/edit/react-oq9isl?file=index.js>

Warning: `value` prop on `input` should not be null

Warning: A component is changing an uncontrolled input of type text to be controlled. Input elements should not switch from uncontrolled to controlled (or vice versa)

6. Props with no value n send to children like <test name/> ==> nothing happens and it assumes as name = true and creates confusion with es6 object {foo} similar to {foo:foo}

7. React put ‘px’ automatically if height given as 10 in styles object ? ==> yes

8. Jsx in browser - Doesn’t work.

Using Html in Js with use of JSX. Write normal html without strings in js file.

React uses Babel to compiles JSX(javascript extension) code to react elements which is in es5 and send it to browser to display

**Render =>**

**9. Render** a **React** element into the DOM in the supplied container and return a reference to the component (or returns null for stateless components). ... If the optional callback is provided, it will be executed after the component is rendered or updated

10. Virtual dom - answer given below somewhere  
Es6 to es5 write es5 with react create class n create element

**11. Const to renderdom** ==> <https://codepen.io/anon/pen/JaPyEO?&editors=0010>

12. IndexRoute, BrowserHistory …

=> **indexRoute** is like home page when no url matches … in Rv16.0 it uses exact attribute to tell router home

React Router (v3) I can accept a server response and use **browserHistory**.push to go to the appropriate response page

React 3 indexroute is similar to exat attr in v4.

In v4, import history from '../history'; so that history.push to go to desired page.

<Provider store={props.store}>

<Router history={browserHistory} routes={routes} />

</Provider>

If sri50 retweets, message reaches all ppl including itisprashant but if prashant retweets it willl reach less followers than sri50 so if follow only sri50 imporves performance instead of following prashant like ppl and getting retweets again and again to loose performance.

Functional componets =>

Functional components have a few 'limitations', which I consider to be their greatest strengths.

1. functional component cannot have a ref assigned to it. While a ref can be a convenient way for a component to 'look up' it's children and communicate with them, my feeling is that this is The Wrong Way to write React. refs encourage a very imperative, almost jquery-like way of writing components, taking us away from the functional, **one-way data flow philosophy for which we chose React in the first place!**

The other big difference is that functional components cannot have state attached to them, which is also a huge advantage

<https://camjackson.net/post/9-things-every-reactjs-beginner-should-know#write-stateless-components>

react children – don’t use props.children.map instead use React.children.map

<Hello name={this.state.name}>children</Hello> (in parent component render)

const sd = React.Children.map(props.children, t=> <div>{t}</div>)

(in child component)

Use above in starter react project in stackblitz

Immutable Immutable js redux saga, observable in redux

Router

Dynamic routing where u can route a component anywhere in app not like static app used to angular ember which got initiated in root module. Before Reactv4 also implemented same.

Nested routes(using match.url in nested route to select from relative url and goes on), responsive routes(using Media query component and switch condition to responsive layout)

FYI, React Router is a sort of a wrapper for HTML5 History API. We can create History.js and implement customHistory.js

<https://medium.freecodecamp.org/you-might-not-need-react-router-38673620f3d>

Try nested Switch

Why jsx needs parent div binding for every component

Sridhar => How you are using react route in idse, how will u change url without router

Router with memory router implemented from idse library

How will you immutate deep object property in setstate or redux

=> use immutable.js or deep clone json.parse(json.stringify(obj))

React scoped stylesheet

4 ways – globally declared,

**as object in inline js** styles so lives within same js **common in react native** or using **radium** lib,

**Modular style** src/ components/ **Gator**/ index.js styles.css **Bite**/ index.js styles.css

## Stylized Components – using **glamorous** lib

const Container = glamorous.div(

{ fontSize: "12pt", margin: "25px auto", padding: "5px"}

)

const Header = glamorous.h1(

{ fontSize: "24pt", fontWeight: "bold"}

)

Last two are good but nothing really come up without library.

What we do in IDSE is, keep modular structure and compile scss using webconfig loader and use as styles.classname and it creates d.ts files instantly

Try button with link to in react => it will create anchor tag inside button

purecomponent – it doesn’t have shouldComponentUpdate lifehook

a.b.c.d.e.t.d.g.e.g = 456

**controlled** component where your form data lives in component state. controlled components support instant field validation (reactive in angular)

An **uncontrolled** component is where your form data is handled by the DOM, instead of inside your React component.(template driven in ng)

You use refs to accomplish this.

react lifehooks

<http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/>

**GetDerivedStatefromProps** is replacement of componentWillMount which returns objects similar to setState in componentwillmount.

Its called after constructor … Its like ngChanges in Angular

**Fiber**, the new React reconciliation algorithm, has the ability to start and stop rendering as needed for performance benefits.

React may start calling componentWillMount at various times whenever it feels like it needs to. So **componentDidMount** called for AJAX requests. componentWillMount used for synchronous action and when api response didn’t come, we have to initalise in contructor or propTypes

There is a common misconception that fetching in **componentWillMount** lets you avoid the first empty rendering state. In practice this was never true because React has always executed render immediately after componentWillMount. If the data is not available by the time componentWillMount fires, the first render will still show a loading state regardless of where you initiate the fetch. This is why moving the fetch to **componentDidMount** has no perceptible effect in the vast majority of cases.

<https://reactjs.org/blog/2018/03/27/update-on-async-rendering.html#examples>

componentWillMount was not useful for one-pass server rendering anyway because it is synchronous so you can’t wait for the data. So if you already have it synchronously, you might as well read it in the constructor.

If you did two rendering passes (which is bad for performance but somewhat works around the issue) then you can keep using UNSAFE\_componentWillMount in the short term. It‘s not safe for async rendering, but your code was already relying on a slow pattern (rendering twice) so it’s better to be explicit about it.

<https://daveceddia.com/watch-out-for-undefined-state/>

<https://stackblitz.com/edit/react-cscdou?file=index.js>

“The end goal of reconciliation is to, in the most efficient way possible, update the UI based on new state” we know certain section of UI isn’t going to change, so no reason to update component and its child so **shouldComponentUpdate** returns false for performance issues.

setState –

This is basically kicking off a process that React calls reconciliation. The reconciliation process is the way React updates the DOM, by making changes to the component based on the change in state. When the request to setState() is triggered, React creates a new tree containing the reactive elements in the component (along with the updated state). This tree is used to figure out how the Search component’s UI should change in response to the state change by comparing it with the elements of the previous tree. React knows which changes to implement and will only update the parts of the DOM where necessary. This is why **React is fast**.

SetState works like it updates the property of object tree by merging obj using Object.assign

The rule of thumb is to never mutate state directly. Always use setState() to change state. Modifying state directly, like the snippet below will not cause the component to re-render.

// do not do this

this.state = {

searchTerm: event.target.value

}

If u want update 3 times

handleIncrement = () => {

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

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

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

}

above method wont work since

The above code snippet is equivalent to:

Object.assign(

{},

{ count: this.state.count + 1 },

{ count: this.state.count + 1 },

{ count: this.state.count + 1 },

)

So it copies same object to destination.

Instead can use updater by accessing previous state

this.setState((prevState) => {

return { count: prevState.count + 3}

})

<https://css-tricks.com/understanding-react-setstate/>

SyntheticEvent, which is React’s cross-browser

redux - actions, reducers, store, connect(mapstatetoprops, mapdispatchtoprops), configstore,

when action is dispatched, all reducers are called and depend on action type , particular state changes and other reducers return same state

Redux thunk:

Before using redux thunk(middleware to handle async actions).

Normally , ajax request are made in **componentDidMount** where will hit the service and do async actions but if u r using redux connect , first thought will be , you will transfer calling api service in actions by calling this.props.fetchdata(url) from componentDidMount with url alone. Then from mapDispatchtoProps provide dispatch to fetchData method and dispatch actions . So thought it will work but it wont Error: Actions must be plain objects. Use custom middleware for async actions. So use redux thunk which will return function instead of just plain object actions

Without [middleware](https://redux.js.org/advanced/middleware), Redux store only supports [synchronous data flow](https://redux.js.org/basics/data-flow)

// Thunk middleware knows how to handle functions.

// It passes the dispatch method as an argument to the function,

// thus making it able to dispatch actions itself.

return function (dispatch) {

// First dispatch: the app state is updated to inform

// that the API call is starting.

dispatch(requestPosts(subreddit))

// The function called by the thunk middleware can return a value,

// that is passed on as the return value of the dispatch method.

// In this case, we return a promise to wait for.

// This is not required by thunk middleware, but it is convenient for us.

return fetch(`https://www.reddit.com/r/${subreddit}.json`)

.then(

response => response.json(),

// Do not use catch, because that will also catch

// any errors in the dispatch and resulting render,

// causing a loop of 'Unexpected batch number' errors.

// <https://github.com/facebook/react/issues/6895>

error => console.log('An error occurred.', error)

)

.then(json =>

// We can dispatch many times!

// Here, we update the app state with the results of the API call.

dispatch(receivePosts(subreddit, json))

)

}

Here only angular uses effects in between component n action.. refer WLL

<http://work.haufegroup.io/intro-redux/>

why ngrx and redux used=>

“When two components need to share state which are like adjacent to each other, I need to lift that state up instead of trying to keep their states in sync.”??????

A component based architecture naturally makes sharing state more difficult. If two components rely on the same state, where should that state live? This was such a popular question that it spurred an entire ecosystem of solutions which eventually ended with Redux. Redux’s solution is to put that shared state in another location called a “store”. Components can then subscribe to any portions of the store they need and can also dispatch “actions” to update the store. React’s solution is to find the nearest parent of both of those components and have that parent manage the shared state, passing it down to the child components as needed. There are pros and cons to both approaches but it’s important to be aware that both solutions exist.

On its own, React is a useful library for creating composable views. However, React doesn't prescribe any specific way of synchronizing data throughout your application. As far as a React component is concerned, data flows down through its children through the props you specify on each element. Some of those props might be functions that update the state one way or another, but how that happens is an open question.

Because React on its own does not focus on application state management, the React community uses libraries like Redux and MobX.

<https://github.com/Microsoft/TypeScript-React-Starter#adding-state-management>

can use custom store manager to store data in state and also in localstorage to fetch when refresh. React newly introduced context APi in 16.4 which will replace redux I think

Axios service

Its like promise based api which is build like http client in angular. Its similar feature to fetch api... get,post,put,delete

Lightweight api ….......import axios from 'axios';

<https://alligator.io/react/axios-react/>

Virtual Dom use

concept of diffing the DOM of the new state with the previous state and only render the difference, which is what ReactJS is doing with **Virtual DOM**.

Manual DOM manipulation is expensive and slow when state changes , have to recreate full DOM which makes page slower in large applications.

So in React, virtual dom create virtual tree objects and match the difference with previous V tree DOM and send updates to actual DOM to update the particular part.

Event delegation,

<https://github.com/Matt-Esch/virtual-dom>

<https://gist.github.com/Raynos/8414846>

<https://calendar.perfplanet.com/2013/diff/>

<https://medium.com/@gethylgeorge/how-virtual-dom-and-diffing-works-in-react-6fc805f9f84e>

Data to Child from parent ways - use callback method from parent and call it with data to send to parent and change data OR dispatch event to store to update. Uniflow directional or context api

Ref - take instance of dom element value to give which submit action or other action performed.. refs are uncontrolled component and can do use it for hurry work code.

React Vs angular

* When doing component tree in pinboard, angular refreshes all components when some state changed so recursion occurs nicely while fetching the children again to draw svg in UI. Wherein case React failed to update recursion since it updates in the particular object property to refresh
* React state changes only if setState triggered I.e doesn’t mutate. Wherein angular updates when we change something.
* Fast since uniflow direction n non mutation
* React transpiles code into javascript and handover to js vm machine to process wherein after js compiled it is handled by angular to parse and its manipulation according to its principle and send to js vm machine. So it’s a very clean process in react.

Why React is different? And design I guess

one-way data flow philosophy for which we chose React in the first place!

<https://camjackson.net/post/9-things-every-reactjs-beginner-should-know>

If you’re new to React, you probably only worked with component classes and instances before. For example, you may declare a Button *component* by creating a class. When the app is running, you may have several *instances* of this component on screen, each with its own properties and local state. This is the **traditional object-oriented UI programming**. Why introduce *elements*?

In this traditional UI model, it is up to you to take care of creating and destroying child component instances. If a Form component wants to render a Button component, it needs to create its instance, and manually keep it up to date with any new information.

This is pseudocode, but it is more or less what you end up with when you write composite UI code that behaves consistently in an object-oriented way using a library like Backbone.

Each component instance has to keep references to its DOM node and to the instances of the children components, and create, update, and destroy them when the time is right. The lines of code grow as the square of the number of possible states of the component, and the parents have direct access to their children component instances, making it hard to decouple them in the future.

**So how is React different**?

**An element is a plain object *describing* a component instance or DOM node and its desired properties.** It contains only information about the component type (for example, a Button), its properties (for example, its color), and any child elements inside it.

An element is not an actual instance. Rather, it is a way to tell React what you *want* to see on the screen. You can’t call any methods on the element. It’s just an immutable description object with two fields: type: (string | ReactClass) and props: Object[1](https://reactjs.org/blog/2015/12/18/react-components-elements-and-instances.html#fn-1).

When an element’s type is a string, it represents a DOM node with that tag name, and props correspond to its attributes. This is what React will render. For example:

{ type: 'button', props: { className: 'button button-blue', children: { type: 'b', props: { children: 'OK!' } } }}

This element is just a way to represent the following HTML as a plain object:

<button class='button button-blue'> <b> OK! </b></button>

What’s important is that both child and parent elements are *just descriptions and not the actual instances*. They don’t refer to anything on the screen when you create them. You can create them and throw them away, and it won’t matter much.

React elements are easy to traverse, don’t need to be parsed, and of course they are much lighter than the actual DOM elements—they’re just objects!

### **Component Elements**

However, the type of an element can also be a function or a class corresponding to a React component:

{ type: Button, props: { color: 'blue', children: 'OK!' }}

This is the core idea of React.

**An element describing a component is also an element, just like an element describing the DOM node. They can be nested and mixed with each other.**

This feature lets you define a DangerButton component as a Button with a specific color property value without worrying about whether Button renders to a DOM <button>, a <div>, or something else entirely:

const DangerButton = ({ children }) => ({ type: Button, props: { color: 'red', children: children }});

More things are there to cover, plZ refer below link for detailed understanding

<https://reactjs.org/blog/2015/12/18/react-components-elements-and-instances.html>

For a React component, props are the input, and an element tree is the output.

**The returned element tree can contain both elements describing DOM nodes, and elements describing other components. This lets you compose independent parts of UI without relying on their internal DOM structure.**

We let React create, update, and destroy instances. We *describe* them with elements we return from the components, and React takes care of managing the instances.

**However, whether functions or classes, fundamentally they are all components to React. They take the props as their input, and return the elements as their output.**

**React Design**

**React takes care of creating an instance for every class component, so you can write components in an object-oriented way with methods and local state, but other than that, instances are not very important in the React’s programming model and are managed by React itself.**

Best practice in react - shouldUpdateComponent, composition rather than inheritance, HOC , thinking in react

Advanced Guides

Performance -

People still talk about them because in practice, they are very hard to implement in regular JavaScript code. What makes React stand out is that all those optimizations happen by default. This makes it hard to shoot yourself in the foot and make your app slow.

The performance cost model of React is also very simple to understand: every setState re-renders the whole sub-tree. If you want to squeeze out performance, call setState as low as possible and use shouldComponentUpdate to prevent re-rendering an large sub-tree.

React without Es6

var createReactClass = require('create-react-class');

class Greeting extends React.Component { // ...}Greeting.defaultProps = { name: 'Mary'};

With createReactClass(), you need to define getDefaultProps() as a function on the passed object:

var Greeting = createReactClass({ getDefaultProps: function() { return { name: 'Mary' }; }, // ...});

With createReactClass(), you have to provide a separate getInitialState method that returns the initial state:

With createReactClass(), this is not necessary because it binds all methods:

Mixins

## **Single-page Application**

A single-page application is an application that loads a single HTML page and all the necessary assets (such as JavaScript and CSS) required for the application to run. Any interactions with the page or subsequent pages do not require a round trip to the server which means the page is not reloaded.

**Though you may build a single-page application in React, it is not a requirement. React can also be used for enhancing small parts of existing websites with additional interactivity. Code written in React can coexist peacefully with markup rendered on the server by something like PHP, or with other client-side libraries. In fact, this is exactly how React is being used at Facebook.**

**why typescript in react when babel available**

<https://medium.com/@amcdnl/react-typescript-%EF%B8%8F-647aa7d054a9>

**Debugging react components and performance:**

<https://building.calibreapp.com/debugging-react-performance-with-react-16-and-chrome-devtools-c90698a522ad>

<https://blog.logrocket.com/using-the-new-streaming-performance-monitor-in-google-chrome-d3019afe95e4>

MISC

“Components don’t necessarily have to correspond to DOM nodes.”

When we first learn React we’re taught that “Components are the building blocks of React. They take in input and return some UI (descriptor)“. Does that mean that every component needs to directly return UI descriptors as we’re typically taught? What if we wanted to have a component render another component (Higher Order Component pattern)? What if we wanted a component to manage some slice of state and then instead of returning a UI descriptor, it returns a function invocation passing in the state (Render Props pattern)? What if we had a component that was in charge of managing sound rather than a visual UI, what would it return? What’s great about React is you don’t **have** to return typical “views” from your components. As long as what eventually gets returned is a React element, null, or false, you’re good.

You can return other components like HOC or render () { return this.props.children(this.someImportantState)}

Or null

**Progressive Disclosure Pattern and mobX**

This web app depicts the "Progressive Disclosure Pattern" using a use case of React Mobile Shopping experience

The user journey (Single Page Checkout)

1. Select a product and the quantity
2. Enter the user contact information
3. Make the payment
4. On successful payment we can finally complete the checkout process

The user state is maintained in the MobX store and as the user progresses further the components are progressively shown based on where the user journey has reached.

Since this is a concept demonstration there are no validations added.

## Why Accessibility?

Web accessibility (also referred to as [a11y](https://en.wiktionary.org/wiki/a11y)) is the design and creation of websites that can be used by everyone. Accessibility support is necessary to allow assistive technology to interpret web pages.

Using fragments

return ( <table> <tr> <Columns /> </tr> </table> );

If columns returns <div><td><td></div> multiple tds in table , it’s a invalid html so wrapping that by React.fragments . Can be used <>dfsdf<> empty string also

/\* The AOT compiler needs the `./` to show that this is local \*/

@import './hero-details-box.css';

<https://www.mockapi.io/docs>

<https://swapi.co/>

<https://openweathermap.org/api>