React Interview Questions & Answers

> Click ⭐ if you like the project. Pull Request are highly appreciated. Follow me @SudheerJonna for technical updates.

Note: This repository is specific to ReactJS. Please check Javascript Interview questions for core javascript questions and DataStructures and Algorithms for DSA related questions or problems.

Table of Contents

▼ Hide/Show table of contents

No.	Questions
8	What are Pure Components?
9	What is state in React?
10	What are props in React?
11	What is the difference between state and props?
12	What is the difference between HTML and React event handling?
13	What are synthetic events in React?
14	What are inline conditional expressions?
15	What is "key" prop and what is the benefit of using it in arrays of elements?
16	What is Virtual DOM?
17	How Virtual DOM works?
18	What is the difference between Shadow DOM and Virtual DOM?
19	What is React Fiber?
20	What is the main goal of React Fiber?
21	What are controlled components?
22	What are uncontrolled components?
23	What is the difference between createElement and cloneElement?
24	What is Lifting State Up in React?
25	What are Higher-Order components?
26	What is children prop?
27	How to write comments in React?
28	What is reconciliation?
29	Does the lazy function support named exports?
30	Why React uses className over class attribute?
31	What are fragments?
32	Why fragments are better than container divs?
33	What are portals in React?
34	What are stateless components?
35	What are stateful components?
36	How to apply validation on props in React?
37	What are the advantages of React?
38	What are the limitations of React?
39	What are the recommended ways for static type checking?

No.	Questions
40	What is the use of react-dom package?
41	What is ReactDOMServer?
42	How to use InnerHtml in React?
43	How to use styles in React?
44	How events are different in React?
45	What is the impact of indexes as keys?
46	How do you conditionally render components?
47	Why we need to be careful when spreading props on DOM elements??
48	How do you memoize a component?
49	How you implement Server-Side Rendering or SSR?
50	How to enable production mode in React?
51	Do Hooks replace render props and higher order components?
52	What is a switching component?
53	What are React Mixins?
54	What are the Pointer Events supported in React?
55	Why should component names start with capital letter?
56	Are custom DOM attributes supported in React v16?
57	How to loop inside JSX?
58	How do you access props in attribute quotes?
59	What is React PropType array with shape?
60	How to conditionally apply class attributes?
61	What is the difference between React and ReactDOM?
62	Why ReactDOM is separated from React?
63	How to use React label element?
64	How to combine multiple inline style objects?
65	How to re-render the view when the browser is resized?
66	How to pretty print JSON with React?
67	Why you can't update props in React?
68	How to focus an input element on page load?
69	How can we find the version of React at runtime in the browser?
70	How to add Google Analytics for react-router?
71	How do you apply vendor prefixes to inline styles in React?

No.	Questions
72	How to import and export components using react and ES6?
73	What are the exceptions on React component naming?
74	Is it possible to use async/await in plain React?
75	What are the common folder structures for React?
76	What are the popular packages for animation?
77	What is the benefit of styles modules?
78	What are the popular React-specific linters?
	React Router
79	What is React Router?
80	How React Router is different from history library?
81	What are the <Router> components of React Router v6?
82	What is the purpose of push and replace methods of history?
83	How do you programmatically navigate using React router v4?
84	How to get query parameters in React Router v4
85	Why you get "Router may have only one child element" warning?
86	How to pass params to history.push method in React Router v4?
87	How to implement default or NotFound page?
88	How to get history on React Router v4?
89	How to perform automatic redirect after login?
	React Internationalization
90	What is React Intl?
91	What are the main features of React Intl?
92	What are the two ways of formatting in React Intl?
93	How to use FormattedMessage as placeholder using React Intl?
94	How to access current locale with React Intl
95	How to format date using React Intl?
	React Testing
96	What is Shallow Renderer in React testing?
97	What is TestRenderer package in React?
98	What is the purpose of ReactTestUtils package?
99	What is Jest?
100	What are the advantages of Jest over Jasmine?

No.	Questions
101	Give a simple example of Jest test case
	React Redux
102	What is Flux?
103	What is Redux?
104	What are the core principles of Redux?
105	What are the downsides of Redux compared to Flux?
106	What is the difference between mapStateToProps() and mapDispatchToProps()?
107	Can I dispatch an action in reducer?
108	How to access Redux store outside a component?
109	What are the drawbacks of MVW pattern
110	Are there any similarities between Redux and RxJS?
111	How to reset state in Redux?
112	What is the difference between React context and React Redux?
113	Why are Redux state functions called reducers?
114	How to make AJAX request in Redux?
115	Should I keep all component's state in Redux store?
116	What is the proper way to access Redux store?
117	What is the difference between component and container in React Redux?
118	What is the purpose of the constants in Redux?
119	What are the different ways to write mapDispatchToProps()?
120	What is the use of the ownProps parameter in mapStateToProps() and mapDispatchToProps()?
121	How to structure Redux top level directories?
122	What is redux-saga?
123	What is the mental model of redux-saga?
124	What are the differences between call and put in redux-saga
125	What is Redux Thunk?
126	What are the differences between redux-saga and redux-thunk
127	What is Redux DevTools?
128	What are the features of Redux DevTools?
129	What are Redux selectors and Why use them?
130	What is Redux Form?
131	What are the main features of Redux Form?

No.	Questions
132	How to add multiple middlewares to Redux?
133	How to set initial state in Redux?
134	How Relay is different from Redux?
135	What is an action in Redux?
	React Native
136	What is the difference between React Native and React?
137	How to test React Native apps?
138	How to do logging in React Native?
139	How to debug your React Native?
	React supported libraries and Integration
140	What is reselect and how it works?
141	What is Flow?
142	What is the difference between Flow and PropTypes?
143	How to use font-awesome icons in React?
144	What is React Dev Tools?
145	Why is DevTools not loading in Chrome for local files?
146	How to use Polymer in React?
147	What are the advantages of React over Vue.js?
148	What is the difference between React and Angular?
149	Why React tab is not showing up in DevTools?
150	What are styled components?
151	Give an example of Styled Components?
152	What is Relay?
	Miscellaneous
153	What are the main features of reselect library?
154	Give an example of reselect usage?
155	Can Redux only be used with React?
156	Do you need to have a particular build tool to use Redux?
157	How Redux Form initialValues get updated from state?
158	How React PropTypes allow different type for one prop?
159	Can I import an SVG file as react component?
160	What is render hijacking in React?

No.	Questions
161	How to pass numbers to React component?
162	Do I need to keep all my state into Redux? Should I ever use react internal state?
163	What is the purpose of registerServiceWorker in React?
164	What is React memo function?
165	What is React lazy function?
166	How to prevent unnecessary updates using setState?
167	How do you render Array, Strings and Numbers in React 16 Version?
168	What are hooks?
169	What rules need to be followed for hooks?
170	How to ensure hooks followed the rules in your project?
171	What are the differences between Flux and Redux?
172	What are the benefits of React Router V4?
173	Can you describe about componentDidCatch lifecycle method signature?
174	In which scenarios do error boundaries not catch errors?
175	What is the behavior of uncaught errors in react 16?
176	What is the proper placement for error boundaries?
177	What is the benefit of component stack trace from error boundary?
178	What are default props?
179	What is the purpose of displayName class property?
180	What is the browser support for react applications?
181	What is code-splitting?
182	What are Keyed Fragments?
183	Does React support all HTML attributes?
184	When component props defaults to true?
185	What is NextJS and major features of it?
186	How do you pass an event handler to a component?
187	How to prevent a function from being called multiple times?
188	How JSX prevents Injection Attacks?
189	How do you update rendered elements?
190	How do you say that props are read only?
191	What are the conditions to safely use the index as a key?
192	Is it keys should be globally unique?

No.	Questions
193	What is the popular choice for form handling?
194	What are the advantages of formik over redux form library?
195	Why do you not required to use inheritance?
196	Can I use web components in react application?
197	What is dynamic import?
198	What are loadable components?
199	What is suspense component?
200	What is route based code splitting?
201	What is the purpose of default value in context?
202	What is diffing algorithm?
203	What are the rules covered by diffing algorithm?
204	When do you need to use refs?
205	Is it prop must be named as render for render props?
206	What are the problems of using render props with pure components?
207	What is windowing technique?
208	How do you print falsy values in JSX?
209	What is the typical use case of portals?
210	How do you set default value for uncontrolled component?
211	What is your favorite React stack?
212	What is the difference between Real DOM and Virtual DOM?
213	How to add Bootstrap to a react application?
214	Can you list down top websites or applications using react as front end framework?
215	Is it recommended to use CSS In JS technique in React?
216	Do I need to rewrite all my class components with hooks?
217	How to fetch data with React Hooks?
218	Is Hooks cover all use cases for classes?
219	What is the stable release for hooks support?
220	Why do we use array destructuring (square brackets notation) in useState?
221	What are the sources used for introducing hooks?
222	How do you access imperative API of web components?
223	What is formik?
224	What are typical middleware choices for handling asynchronous calls in Redux?

No.	Questions
225	Do browsers understand JSX code?
226	Describe about data flow in react?
227	What is MobX?
228	What are the differences between Redux and MobX?
229	Should I learn ES6 before learning ReactJS?
230	What is Concurrent Rendering?
231	What is the difference between async mode and concurrent mode?
232	Can I use javascript urls in react16.9?
233	What is the purpose of eslint plugin for hooks?
234	What is the difference between Imperative and Declarative in React?
235	What are the benefits of using typescript with reactjs?
236	How do you make sure that user remains authenticated on page refresh while using Context API State Management?
237	What are the benefits of new JSX transform?
238	How is the new JSX transform different from old transform?
239	What are React Server components?
240	What is prop drilling?
241	What is the difference between useState and useRef hook?
242	What is a wrapper component
243	What are the differences between useEffect and useLayoutEffect hooks
244	What are the differences between Functional and Class Components
245	What is strict mode in React?
246	What is the benefit of strict mode?
247	Why does strict mode render twice in React?
248	What are the rules of JSX?
249	What is the reason behind multiple JSX tags to be wrapped?
250	How do you prevent mutating array variables?
251	What are capture phase events?
252	How does React updates screen in an application?
253	How does React batch multiple state updates?
254	Is it possible to prevent automatic batching?
255	What is React hydration?

No.	Questions
256	How do you update objects inside state?
257	How do you update nested objects inside state?
258	How do you update arrays inside state?
259	How do you use immer library for state updates?
260	What are the benefits of preventing the direct state mutations?
261	What are the preferred and non-preferred array operations for updating the state?
262	What will happen by defining nested function components?
263	Can I use keys for non-list items?
264	What are the guidelines to be followed for writing reducers?
265	What is useReducer hook? Can you describe its usage?
266	How do you compare useState and useReducer?
267	How does context works using useContext hook?
268	What are the use cases of useContext hook?
269	When to use client and server components?
270	What are the differences between page router and app router in nextjs?

Table of Contents

▼ Hide/Show table of contents

No.	Questions
14	What is the purpose of using super constructor with props argument?
15	How to set state with a dynamic key name?
16	What would be the common mistake of function being called every time the component renders?
17	What are error boundaries in React v16
18	How are error boundaries handled in React v15?
19	What is the purpose of render method of react-dom?
20	What will happen if you use setState in constructor?
21	Is it good to use setState() in componentWillMount() method?
22	What will happen if you use props in initial state?
23	How you use decorators in React?
24	What is CRA and its benefits?
25	What is the lifecycle methods order in mounting?
26	What are the lifecycle methods going to be deprecated in React v16?
27	What is the purpose of getDerivedStateFromProps() lifecycle method?
28	What is the purpose of getSnapshotBeforeUpdate() lifecycle method?
29	What is the recommended way for naming components?
30	What is the recommended ordering of methods in component class?
31	Why we need to pass a function to setState()?
32	Why is isMounted() an anti-pattern and what is the proper solution?
33	What is the difference between constructor and getInitialState?
34	Can you force a component to re-render without calling setState?
35	What is the difference between super() and super(props) in React using ES6 classes?
36	What is the difference between setState and replaceState methods?
37	How to listen to state changes?
38	What is the recommended approach of removing an array element in react state?
39	Is it possible to use React without rendering HTML?
40	What are the possible ways of updating objects in state?
41	What are the approaches to include polyfills in your create-react-app?
42	How to use https instead of http in create-react-app?
43	How to avoid using relative path imports in create-react-app?
44	How to update a component every second?
45	Why is a component constructor called only once?

No.	Questions
46	How to define constants in React?
47	How to programmatically trigger click event in React?
48	How to make AJAX call and In which component lifecycle methods should I make an AJAX call?
49	What are render props?
50	How to dispatch an action on load?
51	How to use connect from React Redux?
52	Whats the purpose of at symbol in the redux connect decorator?
53	How to use TypeScript in create-react-app application?
54	Does the statics object work with ES6 classes in React?
55	Why are inline ref callbacks or functions not recommended?
56	What are HOC factory implementations?
57	How to use class field declarations syntax in React classes?
58	Why do you not need error boundaries for event handlers?
59	What is the difference between try catch block and error boundaries?
60	What is the required method to be defined for a class component?
61	What are the possible return types of render method?
62	What is the main purpose of constructor?
63	Is it mandatory to define constructor for React component?
64	Why should not call setState in componentWillUnmount?
65	What is the purpose of getDerivedStateFromError?
66	What is the methods order when component re-rendered?
67	What are the methods invoked during error handling?
68	What is the purpose of unmountComponentAtNode method?
69	What are the limitations with HOCs?
70	How to debug forwardRefs in DevTools?
71	Is it good to use arrow functions in render methods?
72	How do you say that state updates are merged?
73	How do you pass arguments to an event handler?
74	How to prevent component from rendering?
75	Give an example on How to use context?
76	How do you use contextType?
77	What is a consumer?

No.	Questions
78	How do you solve performance corner cases while using context?
79	What is the purpose of forward ref in HOCs?
80	Is it ref argument available for all functions or class components?
81	Why do you need additional care for component libraries while using forward refs?
82	How to create react class components without ES6?
83	Is it possible to use react without JSX?
84	How do you create HOC using render props?
85	What is react scripts?
86	What are the features of create react app?
87	What is the purpose of renderToNodeStream method?
88	How do you get redux scaffolding using create-react-app?
89	What is state mutation and how to prevent it?

Core React

1. ### What is React?

   React (aka React.js or ReactJS) is an **open-source front-end JavaScript library** that is used for building composable user interfaces, especially for single-page applications. It is used for handling view layer for web and mobile apps based on components in a declarative approach.

   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.

2. ### What is the history behind React evolution?

   The history of ReactJS started in 2010 with the creation of **XHP**. XHP is a PHP extension which improved the syntax of the language such that XML document fragments become valid PHP expressions and the primary purpose was used to create custom and reusable HTML elements.

   The main principle of this extension was to make front-end code easier to understand and to help avoid cross-site scripting attacks. The project was successful to prevent the malicious content submitted by the scrubbing user.

   But there was a different problem with XHP in which dynamic web applications require many roundtrips to the server, and XHP did not solve this problem. Also, the whole UI was re-rendered for small change in the application. Later, the initial prototype of React is created with the name **FaxJ** by Jordan inspired from XHP. Finally after sometime React has been introduced as a new library into JavaScript world.

   Note: JSX comes from the idea of XHP

3. ### What are the major features of React?

   The major features of React are:

   - Uses **JSX** syntax, a syntax extension of JS that allows developers to write HTML in their JS code.

- It uses **Virtual DOM** instead of Real DOM considering that Real DOM manipulations are expensive.
- Supports **server-side rendering** which is useful for Search Engine Optimizations(SEO).
- Follows **Unidirectional or one-way** data flow or data binding.
- Uses **reusable/composable** UI components to develop the view.

4. What is JSX?

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

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

```
export default function App() {
  return <h1 className="greeting">{"Hello, this is a JSX Code!"}</h1>;
}
```

If you don't use JSX syntax then the respective JavaScript code should be written as below,

```
import { createElement } from "react";

export default function App() {
  return createElement(
    "h1",
    { className: "greeting" },
    "Hello, this is a JSX Code!"
  );
}
```

▶ **See Class**

```
class App extends React.Component {
  render() {
    return <h1 className="greeting">{"Hello, this is a JSX Code!"}</h1>;
  }
}
```

Note: JSX is stricter than HTML

5. 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 cannot be mutated.

The JavaScript representation(Without JSX) of React Element would be as follows:

```
const element = React.createElement("div", { id: "login-btn" }, "Login");
```

and this element can be simiplified using JSX

```
<div id="login-btn">Login</div>
```

The above `React.createElement()` function returns an object as below:

```
{
  type: 'div',
  props: {
    children: 'Login',
    id: 'login-btn'
  }
}
```

Finally, this element renders to the DOM using `ReactDOM.render()`.

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 = ({ handleLogin }) => (
  <div id={"login-btn"} onClick={handleLogin}>
    Login
  </div>
);
```

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

```
const Button = ({ handleLogin }) =>
  React.createElement(
    "div",
    { id: "login-btn", onClick: handleLogin },
    "Login"
  );
```

6. How to create components in React?

Components are the building blocks of creating User Interfaces(UI) in React. There are two possible ways to create a component.

1. **Function Components:** This is the simplest way to create a component. Those are pure JavaScript functions that accept props object as the one and only one parameter and return React elements to render the output:

   ```
   function Greeting({ message }) {
     return <h1>{`Hello, ${message}`}</h1>;
   }
   ```

2. **Class Components:** You can also use ES6 class to define a component. The above function component can be written as a class component:

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

7. When to use a Class Component over a Function Component?

After the addition of Hooks(i.e. React 16.8 onwards) it is always recommended to use Function components over Class components in React. Because you could use state, lifecycle methods and other features that were only available in class component present in function component too.

But even there are two reasons to use Class components over Function components.

1. If you need a React functionality whose Function component equivalent is not present yet, like Error Boundaries.
2. In older versions, If the component needs *state or lifecycle methods* then you need to use class component.

So the summary to this question is as follows:

Use Function Components:

- If you don't need state or lifecycle methods, and your component is purely presentational.
- For simplicity, readability, and modern code practices, especially with the use of React Hooks for state and side effects.

Use Class Components:

- If you need to manage state or use lifecycle methods.
- In scenarios where backward compatibility or integration with older code is necessary.

Note: You can also use reusable react error boundary third-party component without writing any class. i.e, No need to use class components for Error boundaries.

The usage of Error boundaries from the above library is quite straight forward.

> ***Note when using react-error-boundary:*** ErrorBoundary is a client component. You can only pass props to it that are serializable or use it in files that have a `"use client";` directive.

```
"use client";

import { ErrorBoundary } from "react-error-boundary";

<ErrorBoundary fallback={<div>Something went wrong</div>}>
  <ExampleApplication />
</ErrorBoundary>;
```

8. What are Pure Components?

Pure components are the components which render the same output for the same state and props. In function components, you can achieve these pure components through memoized `React.memo()` API wrapping around the component. This API prevents unnecessary re-renders by comparing the previous props and new props using shallow comparison. So it will be helpful for performance optimizations.

But at the same time, it won't compare the previous state with the current state because function component itself prevents the unnecessary rendering by default when you set the same state again.

The syntactic representation of memoized components looks like below,

```
const MemoizedComponent = memo(SomeComponent, arePropsEqual?);
```

Below is the example of how child component(i.e., EmployeeProfile) prevents re-renders for the same props passed by parent component(i.e.,EmployeeRegForm).

```
import { memo, useState } from "react";

const EmployeeProfile = memo(function EmployeeProfile({ name, email }) {
  return (
    <>
      <p>Name:{name}</p>
      <p>Email: {email}</p>
    </>
  );
});
export default function EmployeeRegForm() {
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");
  return (
    <>
      <label>
        Name:{" "}
        <input value={name} onChange={(e) => setName(e.target.value)} />
      </label>
      <label>
        Email:{" "}
        <input value={email} onChange={(e) => setEmail(e.target.value)} />
      </label>
      <hr />
      <EmployeeProfile name={name} />
    </>
  );
}
```

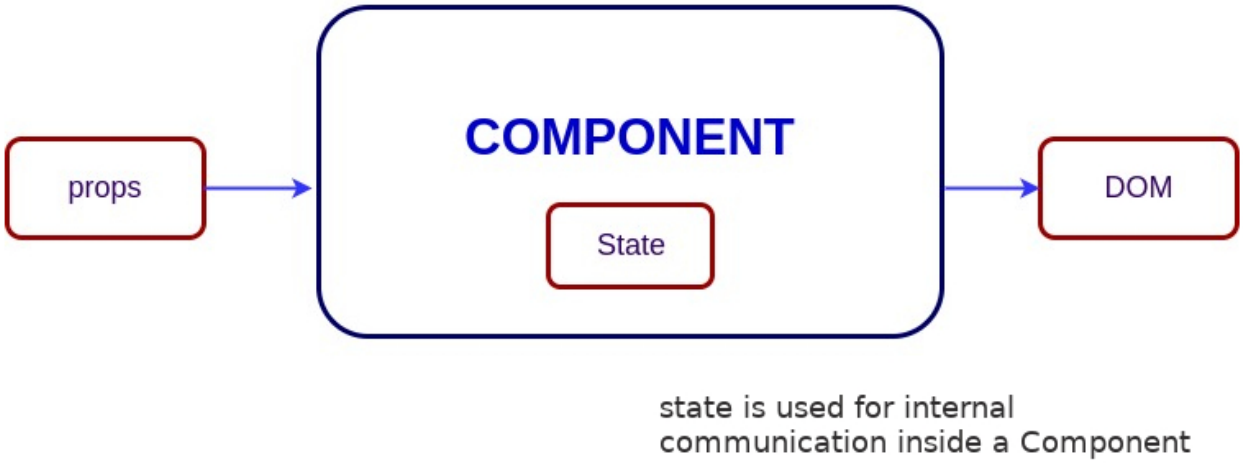
In the above code, the email prop has not been passed to child component. So there won't be any re-renders for email prop change.

In class components, the components extending *`React.PureComponent`* instead of *`React.Component`* become the pure components. When props or state changes, *PureComponent* will do a shallow comparison on both props and state by invoking `shouldComponentUpdate()` lifecycle method.

Note: `React.memo()` is a higher-order component.

9. 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. The important point is whenever the state object changes, the component re-renders. It is always recommended to make our state as simple as possible and minimize the number of stateful components.



Let's take an example of **User** component with `message` state. Here, **useState** hook has been used to add state to the User component and it returns an array with current state and function to update it.

```
import { useState } from "react";

function User() {
  const [message, setMessage] = useState("Welcome to React world");

  return (
    <div>
      <h1>{message}</h1>
    </div>
  );
}
```

Whenever React calls your component or access `useState` hook, it gives you a snapshot of the state for that particular render.

▶ **See Class**

```
import React from "react";
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>
    );
  }
}
```

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 till the owner component decides to pass it.

10. 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 similar to HTML-tag attributes. Here, the data is passed down from a parent component to a child component.

The primary purpose of props in React is to provide following component functionality:

1. Pass custom data to your component.
2. Trigger state changes.
3. Use via `this.props.reactProp` inside component's `render()` method.

For example, let us create an element with `reactProp` property:

```
<Element reactProp={"1"} />
```

This `reactProp` (or whatever you came up with) attribute 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;
```

For example, the usage of props in function component looks like below:

```
import React from "react";
import ReactDOM from "react-dom";

const ChildComponent = (props) => {
  return (
    <div>
      <p>{props.name}</p>
      <p>{props.age}</p>
      <p>{props.gender}</p>
    </div>
  );
};
```

```
  const ParentComponent = () => {
    return (
      <div>
        <ChildComponent name="John" age="30" gender="male" />
        <ChildComponent name="Mary" age="25" geneder="female" />
      </div>
    );
  };
```

The properties from props object can be accessed directly using destructing feature from ES6 (ECMAScript 2015). It is also possible to fallback to default value when the prop value is not specified. The above child component can be simplified like below.

```
const ChildComponent = ({ name, age, gender = "male" }) => {
  return (
    <div>
      <p>{name}</p>
      <p>{age}</p>
      <p>{gender}</p>
    </div>
  );
};
```

Note: The default value won't be used if you pass `null` or `0` value. i.e, default value is only used if the prop value is missed or `undefined` value has been passed.

▶ **See Class**

```
import React from "react";
import ReactDOM from "react-dom";

class ChildComponent extends React.Component {
  render() {
    return (
      <div>
        <p>{this.props.name}</p>
        <p>{this.props.age}</p>
        <p>{this.props.gender}</p>
      </div>
    );
  }
}

class ParentComponent extends React.Component {
  render() {
    return (
      <div>
        <ChildComponent name="John" age="30" gender="male" />
        <ChildComponent name="Mary" age="25" gender="female" />
      </div>
    );
  }
}
```

11. What is the difference between state and props?

    In React, both `state` and `props` are plain JavaScript objects and used to manage the data of a component, but they are used in different ways and have different characteristics.

    The `state` entity is managed by the component itself and can be updated using the setter(`setState()` for class components) function. Unlike props, state can be modified by the component and is used to manage the internal state of the component. i.e, state acts as a component's memory. Moreover, changes in the state trigger a re-render of the component and its children. The components cannot become reusable with the usage of state alone.

    On the otherhand, `props` (short for "properties") are passed to a component by its parent component and are `read-only`, meaning that they cannot be modified by the own component itself. i.e, props acts as arguments for a function. Also, props can be used to configure the behavior of a component and to pass data between components. The components become reusable with the usage of props.

12. 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()"></button>
       ```

       Whereas in React it follows *camelCase* convention:

       ```
       <button onClick={activateLasers}>
       ```

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

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

       Whereas in React you must call `preventDefault()` explicitly:

       ```
       function handleClick(event) {
         event.preventDefault();
         console.log("The link was clicked.");
       }
       ```

    3. 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)

13. What are synthetic events in React?

`SyntheticEvent` is a cross-browser wrapper around the browser's native event. Its API is same as the browser's native event, including `stopPropagation()` and `preventDefault()`, except the events work identically across all browsers. The native events can be accessed directly from synthetic events using `nativeEvent` attribute.

Let's take an example of BookStore title search component with the ability to get all native event properties

```
function BookStore() {
  function handleTitleChange(e) {
    console.log("The new title is:", e.target.value);
    // 'e' represents synthetic event
    const nativeEvent = e.nativeEvent;
    console.log(nativeEvent);
    e.stopPropagation();
    e.preventDefault();
  }

  return <input name="title" onChange={handleTitleChange} />;
}
```

14. 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>
  );
}
```

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

A `key` is a special attribute you **should** include when mapping over arrays to render data. *Key* prop helps React identify which items have changed, are added, or are removed.

Keys should be unique among its siblings. 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.
4. The key attribute accepts either string or number and internally convert it as string type.
5. Don't generate the key on the fly something like `key={Math.random()}`. Because the keys will never match up between re-renders and DOM created everytime.

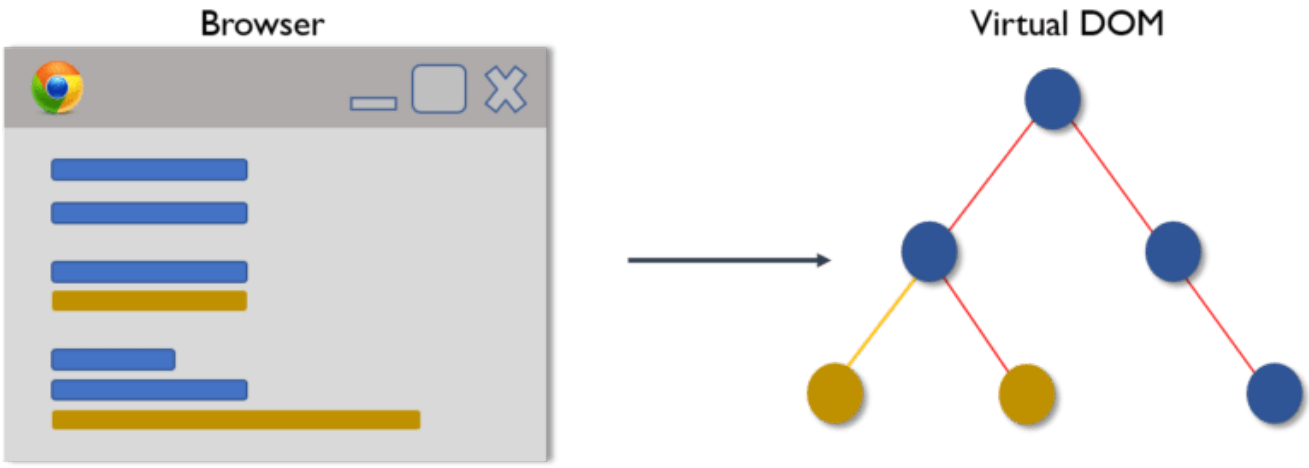
16. What is Virtual DOM?

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*.

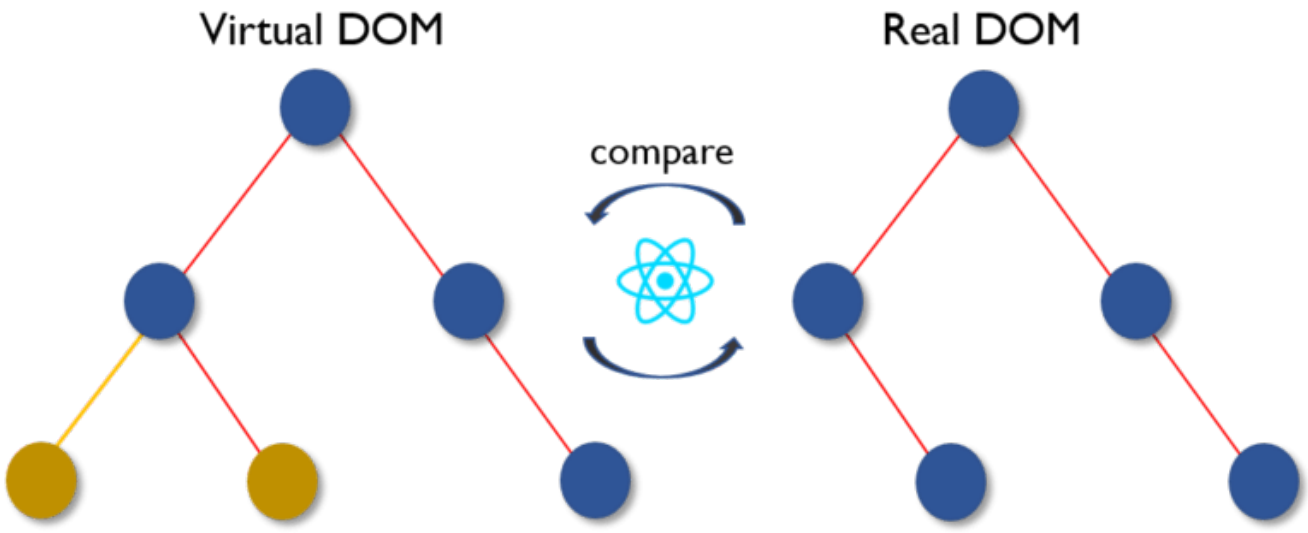
17. How Virtual DOM works?

The *Virtual DOM* works in three simple steps.

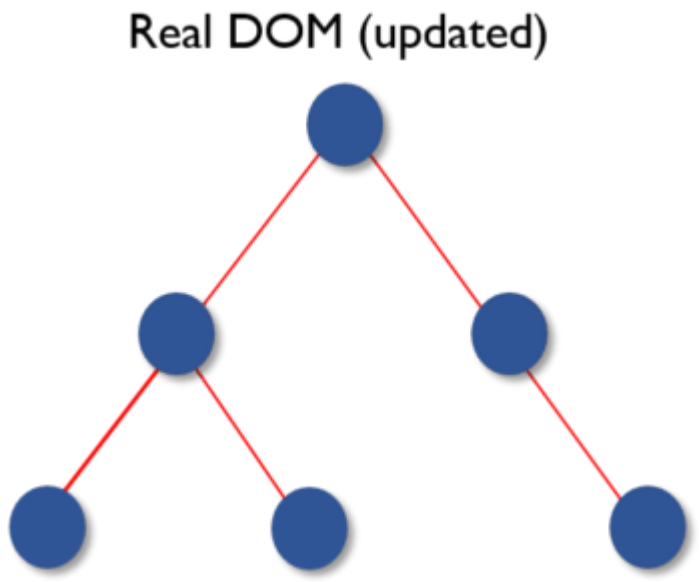
1. Whenever any underlying data changes, the entire UI is re-rendered in Virtual DOM representation.



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



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



18. 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.

19. 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.

20. 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.

21. 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. That means, the displayed data is always in sync with the state of the component.

The controlled components will be implemented using the below steps,

1. Initialize the state using use state hooks in function components or inside constructor for class components.
2. Set the value of the form element to the respective state variable.
3. Create an event handler to handle the user input changes through useState updater function or setState from class component.
4. Attach the above event handler to form elements change or click events

For example, the name input field updates the user name using `handleChange` event handler as below,

```
import React, { useState } from "react";

function UserProfile() {
  const [username, setUsername] = useState("");

  const handleChange = (e) => {
    setUsername(e.target.value);
  };

  return (
    <form>
      <label>
        Name:
        <input type="text" value={username} onChange={handleChange} />
      </label>
    </form>
  );
}
```

22. 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.

The uncontrolled components will be implemented using the below steps,

1. Create a ref using useRef react hook in function component or `React.createRef()` in class based component.
2. Attach this ref to the form element.
3. The form element value can be accessed directly through `ref` in event handlers or `componentDidMount` for class components

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

```
import React, { useRef } from "react";

function UserProfile() {
  const usernameRef = useRef(null);

  const handleSubmit = (event) => {
    event.preventDefault();
    console.log("The submitted username is: " + usernameRef.current.value);
  };
```

```
    return (
      <form onSubmit={handleSubmit}>
        <label>
          Username:
          <input type="text" ref={usernameRef} />
        </label>
        <button type="submit">Submit</button>
      </form>
    );
  }
```

In most cases, it's recommend to use controlled components to implement forms. In a controlled component, form data is handled by a React component. The alternative is uncontrolled components, where form data is handled by the DOM itself.

▶ **See Class**

```
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>
    );
  }
}
```

23. 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.

24. 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.

25. 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.

26. What is children prop?

Children is a prop that allows 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.

A simple usage of children prop looks as below,

```
function MyDiv({ children }){
    return (
      <div>
        {children}
      </div>;
    );
}

export default function Greeting() {
  return (
    <MyDiv>
      <span>{"Hello"}</span>
      <span>{"World"}</span>
    </MyDiv>
  );
}
```

▶ **See Class**

```
const MyDiv = React.createClass({
  render: function () {
    return <div>{this.props.children}</div>;
  },
});

ReactDOM.render(
```

```
  <MyDiv>
    <span>{"Hello"}</span>
    <span>{"World"}</span>
  </MyDiv>,
  node
);
```

Note: There are several methods available in the legacy React API to work with this prop. These include `React.Children.map`, `React.Children.forEach`, `React.Children.count`, `React.Children.only`, `React.Children.toArray`.

27. 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>
```

28. What is reconciliation?

`Reconciliation` is the process through which React updates the Browser DOM and makes React work faster. React use a `diffing algorithm` so that component updates are predictable and faster. React would first calculate the difference between the `real DOM` and the copy of DOM `(Virtual DOM)` when there's an update of components.
React stores a copy of Browser DOM which is called `Virtual DOM`. When we make changes or add data, React creates a new Virtual DOM and compares it with the previous one. This comparison is done by `Diffing Algorithm`.
Now React compares the Virtual DOM with Real DOM. It finds out the changed nodes and updates only the changed nodes in Real DOM leaving the rest nodes as it is. This process is called *Reconciliation*.

29. Does the lazy function support 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"));
```

30. Why React uses `className` over `class` attribute?

The attribute names written in JSX turned into keys of JavaScript objects and the JavaScript names cannot contain dashes or reserved words, it is recommended to use camelCase wherever applicable in JSX code. The attribute `class` is a keyword in JavaScript, and JSX is an extension of JavaScript. That's the principle reason why React uses `className` instead of `class`. Pass a string as the `className` prop.

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

31. What are fragments?

It's a common pattern or practice in React for a component to return multiple elements. *Fragments* let you group a list of children without adding extra nodes to the DOM.
You need to use either `<Fragment>` or a shorter syntax having empty tag (`<></>`).

Below is the example of how to use fragment inside *Story* component.

```
function Story({ title, description, date }) {
  return (
    <Fragment>
      <h2>{title}</h2>
      <p>{description}</p>
      <p>{date}</p>
    </Fragment>
  );
}
```

It is also possible to render list of fragments inside a loop with the mandatory **key** attribute supplied.

```
function StoryBook() {
  return stories.map((story) => (
    <Fragment key={story.id}>
      <h2>{story.title}</h2>
      <p>{story.description}</p>
      <p>{story.date}</p>
    </Fragment>
  ));
}
```

Usually, you don't need to use `<Fragment>` until there is a need of *key* attribute. The usage of shorter syntax looks like below.

```
function Story({ title, description, date }) {
  return (
    <>
      <h2>{title}</h2>
      <p>{description}</p>
      <p>{date}</p>
    </>
  );
}
```

32. Why fragments are better than container divs?

Below are the list of reasons to prefer fragments over container DOM elements,

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.

33. 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. When using
CSS transform in a component, its descendant elements should not use fixed positioning, otherwise the layout will blow up.

```
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.

34. What are stateless components?

If the behaviour of a component 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.

35. 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 either function components with hooks or *class components*.

Let's take an example of function stateful component which update the state based on click event,

```
import React, {useState} from 'react';

const App = (props) => {
const [count, setCount] = useState(0);
handleIncrement() {
  setCount(count+1);
}

return (
  <>
    <button onClick={handleIncrement}>Increment</button>
    <span>Counter: {count}</span>
  </>
  )
}
```

▶ **See Class**

```
class App extends Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

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

  render() {
    <>
      <button onClick={() => this.handleIncrement}>Increment</button>
      <span>Count: {count}</span>
    </>;
  }
}
```

36. 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({ name, age }) {
  return (
    <>
      <h1>{`Welcome, ${name}`}</h1>
      <h2>{`Age, ${age}`}</h2>
    </>
  );
}
```

```
User.propTypes = {
  name: PropTypes.string.isRequired,
  age: PropTypes.number.isRequired,
};
```

37. 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.

38. 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.

39. 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.

40. 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()`

41. 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();
});
```

42. 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()} />;
}
```

43. 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`).

44. 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.

45. 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 and creates confusing bugs in the application.

```
{
  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} />);
}
```

Note: If you don't specify `key` prop at all, React will use index as a key's value while iterating over an array of data.

46. 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>
);
```

47. 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>
);
```

48. 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);
```

49. 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.

50. 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.

51. 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.

52. 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,
};
```

53. 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],
  // ...
});
```

54. 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`

55. 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 an unrecognized tag. This convention is because only HTML elements and SVG tags can begin with a lowercase letter.

```
function SomeComponent {
  // Code goes here
}
```

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

```
function myComponent {
  render() {
    return <div />;
```

```
  }
}

export default myComponent;
```

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

```
import MyComponent from "./myComponent";
```

56. 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.

57. 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*.

58. 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}`} />
```

59. 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,
};
```

60. 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'}`}>
```

61. 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()`.

62. 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.

63. 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'} />
```

64. 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>
```

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

You can use the `useState` hook to manage the width and height state variables, and the `useEffect` hook to add and remove the `resize` event listener. The `[]` dependency array passed to useEffect ensures that the effect only runs once (on mount) and not on every re-render.

```
import React, { useState, useEffect } from "react";
function WindowDimensions() {
  const [dimensions, setDimensions] = useState({
    width: window.innerWidth,
    height: window.innerHeight,
  });

  useEffect(() => {
    function handleResize() {
      setDimensions({
        width: window.innerWidth,
        height: window.innerHeight,
      });
    }
    window.addEventListener("resize", handleResize);
    return () => window.removeEventListener("resize", handleResize);
  }, []);

  return (
    <span>
      {dimensions.width} x {dimensions.height}
    </span>
  );
}
```

▶

Using Class Component

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>
    );
  }
}
```

66. 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 };

function User {
    return <pre>{JSON.stringify(data, null, 2)}</pre>;
}

const container = createRoot(document.getElementById("container"));

container.render(<User />);
```

▶ **See Class**

```
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"));
```

67. Why you can't update props in React?

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

68. How to focus an input element on page load?

You need to use `useEffect` hook to set focus on input field during page load time for functional component.

```
import React, { useEffect, useRef } from "react";

const App = () => {
  const inputElRef = useRef(null);

  useEffect(() => {
    inputElRef.current.focus();
  }, []);

  return (
    <div>
      <input defaultValue={"Won't focus"} />
      <input ref={inputElRef} defaultValue={"Will focus"} />
    </div>
  );
};

ReactDOM.render(<App />, document.getElementById("app"));
```

▶ **See Class**

```
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"));
```

69. 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")
);
```

70. 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);
});
```

71. 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
  }}
/>
```

72. How to import and export components using React and ES6?

You should use default for exporting the components

```
import User from "user";

export default function MyProfile {
    return <User type="customer">//...</User>;
}
```

▶ **See Class**

export default class MyProfile extends React.Component {
render() {
return //...;
}
}

```
</p>
</details>

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.
```

73. What are the exceptions on React component naming?

The component names should start with an uppercase letter but there are few exceptions to 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)`
    )
}
```

74. 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.

75. 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
```

2. **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
```

76. What are the popular packages for animation?

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

77. 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";
```

78. 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.

React Router

79. 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.

80. 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.

81. What are the `<Router>` components of React Router v6?

React Router v6 provides below 4 `<Router>` components:

1. `<BrowserRouter>`:Uses the HTML5 history API for standard web apps.
2. `<HashRouter>`:Uses hash-based routing for static servers.
3. `<MemoryRouter>`:Uses in-memory routing for testing and non-browser environments.
4. `<StaticRouter>`:Provides static routing for server-side rendering (SSR).

The above components will create *browser*, *hash*, *memory* and *static* history instances. React Router v6 makes the properties and methods of the `history` instance associated with your router available through the context in the `router` object.

82. 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.

83. 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>
));
```

2. **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>
    )}
  />
);
```

3. **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,
    }),
  };
```

84. 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.

85. 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>
```

86. 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.

87. 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>
```

88. 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 */
   });
   ```

2. You should use the `<Router>` component instead of built-in routers. Import the above `history.js` inside `index.js` file:

   ```
   import { Router } from "react-router-dom";
   import history from "./history";
   import App from "./App";

   ReactDOM.render(
     <Router history={history}>
       <App />
     </Router>,
     holder
   );
   ```

3. You can also use push method of `history` object similar to built-in history object:

   ```
   // some-other-file.js
   import history from "./history";
   ```

```
history.push("/go-here");
```

89. 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 { Redirect } from "react-router";

export default function Login {
    if (this.state.isLoggedIn === true) {
      return <Redirect to="/your/redirect/page" />;
    } else {
      return <div>{"Login Please"}</div>;
    }
}
```

▶ **See Class**

```
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

90. What is React Intl?

The *React Intl* library makes internationalization 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.

91. 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.

92. 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:**

```
<FormattedMessage
  id={"account"}
  defaultMessage={"The amount is less than minimum balance."}
/>
```

2. **Using an API:**

```
const messages = defineMessages({
  accountMessage: {
    id: "account",
    defaultMessage: "The amount is less than minimum balance.",
  },
});

formatMessage(messages.accountMessage);
```

93. 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);
```

94. 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);
```

95. 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

96. 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>,
]);
```

97. 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' ]
// }
```

98. 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.

99. 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.

100. 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.

101. 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)
```

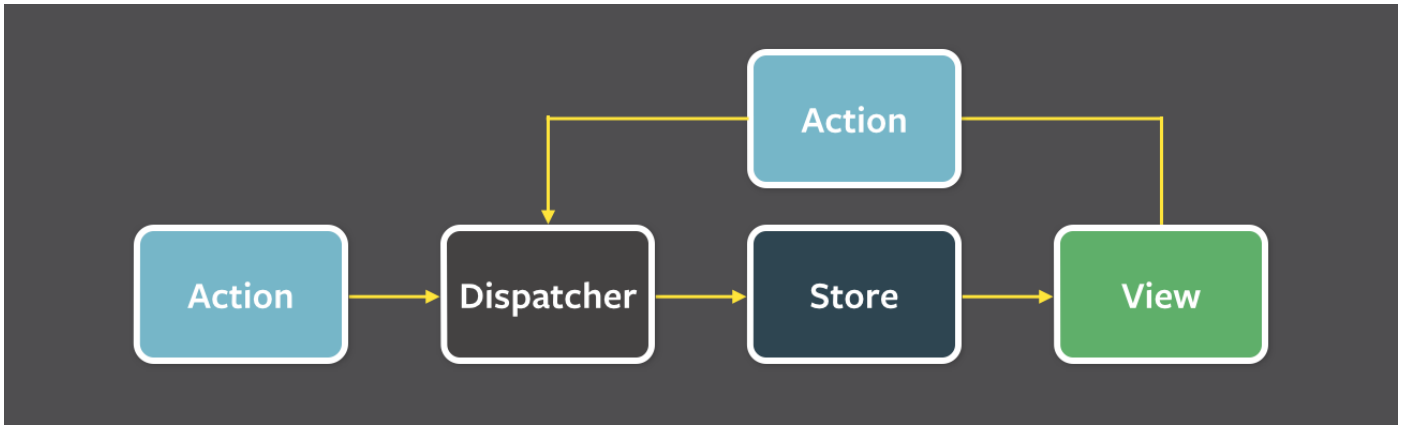
React Redux

102. What is flux?

Flux is an *application design paradigm* used as a replacement for the more traditional MVC pattern. It is not a framework or a library but a new kind of architecture that complements React and the concept of Unidirectional

Data Flow. Facebook uses this pattern internally when working with React.

The workflow between dispatcher, stores and views components with distinct inputs and outputs as follows:



103. What is Redux?

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

104. What are the core principles of Redux?

Redux follows three fundamental principles:

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

105. What are the downsides of Redux compared to Flux?

Instead of saying downsides we can say that there are few compromises of using Redux over Flux. Those are as follows:

1. **You will need to learn to avoid mutations:** Flux is un-opinionated about mutating data, but Redux doesn't like mutations and many packages complementary to Redux assume you never mutate the state. You can enforce this with dev-only packages like `redux-immutable-state-invariant`, Immutable.js, or instructing your team to write non-mutating code.
2. **You're going to have to carefully pick your packages:** While Flux explicitly doesn't try to solve problems such as undo/redo, persistence, or forms, Redux has extension points such as middleware and store enhancers, and it has spawned a rich ecosystem.
3. **There is no nice Flow integration yet:** Flux currently lets you do very impressive static type checks which Redux doesn't support yet.

106. What is the difference between `mapStateToProps()` and `mapDispatchToProps()`?

`mapStateToProps()` is a utility which helps your component get updated state (which is updated by some other components):

```
const mapStateToProps = (state) => {
  return {
    todos: getVisibleTodos(state.todos, state.visibilityFilter),
  };
};
```

`mapDispatchToProps()` is a utility which will help your component to fire an action event (dispatching action which may cause change of application state):

```
const mapDispatchToProps = (dispatch) => {
  return {
    onTodoClick: (id) => {
      dispatch(toggleTodo(id));
    },
  };
};
```

It is recommended to always use the "object shorthand" form for the `mapDispatchToProps`.

Redux wraps it in another function that looks like (...args) => dispatch(onTodoClick(...args)), and pass that wrapper function as a prop to your component.

```
const mapDispatchToProps = {
  onTodoClick,
};
```

107. Can I dispatch an action in reducer?

Dispatching an action within a reducer is an **anti-pattern**. Your reducer should be *without side effects*, simply digesting the action payload and returning a new state object. Adding listeners and dispatching actions within the reducer can lead to chained actions and other side effects.

108. How to access Redux store outside a component?

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

```
store = createStore(myReducer);

export default store;
```

109. What are the drawbacks of MVW pattern?

1. DOM manipulation is very expensive which causes applications to behave slow and inefficient.
2. Due to circular dependencies, a complicated model was created around models and views.
3. Lot of data changes happens for collaborative applications(like Google Docs).
4. No way to do undo (travel back in time) easily without adding so much extra code.

110. ## Are there any similarities between Redux and RxJS?

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

Redux is a tool for managing state throughout the application. It is usually used as an architecture for UIs. Think of it as an alternative to (half of) Angular. RxJS is a reactive programming library. It is usually used as a tool to accomplish asynchronous tasks in JavaScript. Think of it as an alternative to Promises. Redux uses the Reactive paradigm because the Store is reactive. The Store observes actions from a distance, and changes itself. RxJS also uses the Reactive paradigm, but instead of being an architecture, it gives you basic building blocks, Observables, to accomplish this pattern.

111. ## How to reset state in Redux?

You need to write a *root reducer* in your application which delegate handling the action to the reducer generated by `combineReducers()`.

For example, let us take `rootReducer()` to return the initial state after `USER_LOGOUT` action. As we know, reducers are supposed to return the initial state when they are called with `undefined` as the first argument, no matter the action.

```
const appReducer = combineReducers({
  /* your app's top-level reducers */
});

const rootReducer = (state, action) => {
  if (action.type === "USER_LOGOUT") {
    state = undefined;
  }

  return appReducer(state, action);
};
```

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

```
const appReducer = combineReducers({
  /* your app's top-level reducers */
});

const rootReducer = (state, action) => {
  if (action.type === "USER_LOGOUT") {
    Object.keys(state).forEach((key) => {
      storage.removeItem(`persist:${key}`);
    });

    state = undefined;
  }

  return appReducer(state, action);
};
```

112. What is the difference between React context and React Redux?

You can use **Context** in your application directly and is going to be great for passing down data to deeply nested components which what it was designed for.

Whereas **Redux** is much more powerful and provides a large number of features that the Context API doesn't provide. Also, React Redux uses context internally but it doesn't expose this fact in the public API.

113. Why are Redux state functions called reducers?

Reducers always return the accumulation of the state (based on all previous and current actions). Therefore, they act as a reducer of state. Each time a Redux reducer is called, the state and action are passed as parameters. This state is then reduced (or accumulated) based on the action, and then the next state is returned. You could *reduce* a collection of actions and an initial state (of the store) on which to perform these actions to get the resulting final state.

114. How to make AJAX request in Redux?

You can use `redux-thunk` middleware which allows you to define async actions.

Let's take an example of fetching specific account as an AJAX call using *fetch API*:

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

function setAccount(data) {
  return { type: "SET_Account", data: data };
}
```

115. Should I keep all component's state in Redux store?

Keep your data in the Redux store, and the UI related state internally in the component.

116. What is the proper way to access Redux store?

The best way to access your store in a component is to use the `connect()` function, that creates a new component that wraps around your existing one. This pattern is called *Higher-Order Components*, and is generally the preferred way of extending a component's functionality in React. This allows you to map state and action creators to your component, and have them passed in automatically as your store updates.

Let's take an example of `<FilterLink>` component using connect:

```
import { connect } from "react-redux";
import { setVisibilityFilter } from "../actions";
import Link from "../components/Link";

const mapStateToProps = (state, ownProps) => ({
  active: ownProps.filter === state.visibilityFilter,
});

const mapDispatchToProps = (dispatch, ownProps) => ({
  onClick: () => dispatch(setVisibilityFilter(ownProps.filter)),
});

const FilterLink = connect(mapStateToProps, mapDispatchToProps)(Link);

export default FilterLink;
```

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

```
function MyComponent {
  someMethod() {
    doSomethingWith(this.context.store);
  }
}
```

117. What is the difference between component and container in React Redux?

Component is a class or function component that describes the presentational part of your application.

Container is an informal term for a component that is connected to a Redux store. Containers *subscribe* to Redux state updates and *dispatch* actions, and they usually don't render DOM elements; they delegate rendering to presentational child components.

118. What is the purpose of the constants in Redux?

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

Normally we will save them in a single file (`constants.js` or `actionTypes.js`).

```
export const ADD_TODO = "ADD_TODO";
export const DELETE_TODO = "DELETE_TODO";
export const EDIT_TODO = "EDIT_TODO";
export const COMPLETE_TODO = "COMPLETE_TODO";
export const COMPLETE_ALL = "COMPLETE_ALL";
export const CLEAR_COMPLETED = "CLEAR_COMPLETED";
```

In Redux, you use them in two places:

1. **During action creation:**

   Let's take `actions.js`:

```
import { ADD_TODO } from "./actionTypes";

export function addTodo(text) {
  return { type: ADD_TODO, text };
}
```

2. **In reducers:**

   Let's create `reducer.js`:

```
import { ADD_TODO } from "./actionTypes";

export default (state = [], action) => {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state,
        {
          text: action.text,
          completed: false,
        },
      ];
    default:
      return state;
  }
};
```

119. What are the different ways to write `mapDispatchToProps()`?

There are a few ways of binding *action creators* to `dispatch()` in `mapDispatchToProps()`.

Below are the possible options:

```
const mapDispatchToProps = (dispatch) => ({
  action: () => dispatch(action()),
});
```

```
const mapDispatchToProps = (dispatch) => ({
  action: bindActionCreators(action, dispatch),
});
```

```
const mapDispatchToProps = { action };
```

The third option is just a shorthand for the first one.

120. What is the use of the `ownProps` parameter in `mapStateToProps()` and `mapDispatchToProps()`?

If the `ownProps` parameter is specified, React Redux will pass the props that were passed to the component into your *connect* functions. So, if you use a connected component:

```
import ConnectedComponent from "./containers/ConnectedComponent";

<ConnectedComponent user={"john"} />;
```

The `ownProps` inside your `mapStateToProps()` and `mapDispatchToProps()` functions will be an object:

```
{
  user: "john";
}
```

You can use this object to decide what to return from those functions.

121. How to structure Redux top level directories?

Most of the applications has several top-level directories as below:

1. **Components**: Used for *dumb* components unaware of Redux.
2. **Containers**: Used for *smart* components connected to Redux.
3. **Actions**: Used for all action creators, where file names correspond to part of the app.
4. **Reducers**: Used for all reducers, where files name correspond to state key.
5. **Store**: Used for store initialization.

This structure works well for small and medium size apps.

122. What is redux-saga?

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

It is available in NPM:

```
$ npm install --save redux-saga
```

123. What is the mental model of redux-saga?

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

124. What are the differences between `call()` and `put()` in redux-saga?

Both `call()` and `put()` are effect creator functions. `call()` function is used to create effect description, which instructs middleware to call the promise. `put()` function creates an effect, which instructs middleware to dispatch an action to the store.

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

```
function* fetchUserSaga(action) {
  // `call` function accepts rest arguments, which will be passed to `api.fetchUser`
function.
  // Instructing middleware to call promise, it resolved value will be assigned to
`userData` variable
  const userData = yield call(api.fetchUser, action.userId);

  // Instructing middleware to dispatch corresponding action.
  yield put({
    type: "FETCH_USER_SUCCESS",
    userData,
  });
}
```

125. What is Redux Thunk?

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

126. What are the differences between `redux-saga` and `redux-thunk`?

Both *Redux Thunk* and *Redux Saga* take care of dealing with side effects. In most of the scenarios, Thunk uses *Promises* to deal with them, whereas Saga uses *Generators*. Thunk is simple to use and Promises are familiar to many developers, Sagas/Generators are more powerful but you will need to learn them. But both middleware can coexist, so you can start with Thunks and introduce Sagas when/if you need them.

127. What is Redux DevTools?

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

128. What are the features of Redux DevTools?

Some of the main features of Redux DevTools are below,

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

129. What are Redux selectors and why use them?

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

For example, to get user details from the state:

```
const getUserData = (state) => state.user.data;
```

These selectors have two main benefits,

1. The selector can compute derived data, allowing Redux to store the minimal possible state
2. The selector is not recomputed unless one of its arguments changes

130. What is Redux Form?

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

131. What are the main features of Redux Form?

Some of the main features of Redux Form are:

1. Field values persistence via Redux store.
2. Validation (sync/async) and submission.
3. Formatting, parsing and normalization of field values.

132. How to add multiple middlewares to Redux?

You can use `applyMiddleware()`.

For example, you can add `redux-thunk` and `logger` passing them as arguments to `applyMiddleware()`:

```
import { createStore, applyMiddleware } from "redux";
const createStoreWithMiddleware = applyMiddleware(
  ReduxThunk,
  logger
)(createStore);
```

133. How to set initial state in Redux?

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

```
const rootReducer = combineReducers({
  todos: todos,
  visibilityFilter: visibilityFilter,
});

const initialState = {
  todos: [{ id: 123, name: "example", completed: false }],
};

const store = createStore(rootReducer, initialState);
```

134. How Relay is different from Redux?

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

135. What is an action in Redux?

Actions are plain JavaScript objects or payloads of information that send data from your application to your store. They are the only source of information for the store. Actions must have a type property that indicates the type of action being performed.

For example, let's take an action which represents adding a new todo item:

```
{
  type: ADD_TODO,
  text: 'Add todo item'
}
```

React Native

136. What is the difference between React Native and React?

React is a JavaScript library, supporting both front end web and being run on the server, for building user interfaces and web applications.

React Native is a mobile framework that compiles to native app components, allowing you to build native mobile applications (iOS, Android, and Windows) in JavaScript that allows you to use React to build your components, and implements React under the hood.

137. How to test React Native apps?

React Native can be tested only in mobile simulators like iOS and Android. You can run the app in your mobile using expo app (https://expo.io) Where it syncs using QR code, your mobile and computer should be in same wireless network.

138. How to do logging in React Native?

You can use `console.log`, `console.warn`, etc. As of React Native v0.29 you can simply run the following to see logs in the console:

```
$ react-native log-ios
$ react-native log-android
```

139. How to debug your React Native?

Follow the below steps to debug React Native app:

1. Run your application in the iOS simulator.

2. Press `Command + D` and a webpage should open up at `http://localhost:8081/debugger-ui`.
3. Enable *Pause On Caught Exceptions* for a better debugging experience.
4. Press `Command + Option + I` to open the Chrome Developer tools, or open it via `View` -> `Developer` -> `Developer Tools`.
5. You should now be able to debug as you normally would.

React supported libraries & Integration

140. What is reselect and how it works?

Reselect is a **selector library** (for Redux) which uses *memoization* concept. It was originally written to compute derived data from Redux-like applications state, but it can't be tied to any architecture or library.

Reselect keeps a copy of the last inputs/outputs of the last call, and recomputes the result only if one of the inputs changes. If the same inputs are provided twice in a row, Reselect returns the cached output. It's memoization and cache are fully customizable.

141. What is Flow?

Flow is a *static type checker* designed to find type errors in JavaScript. Flow types can express much more fine-grained distinctions than traditional type systems. For example, Flow helps you catch errors involving `null`, unlike most type systems.

142. What is the difference between Flow and PropTypes?

Flow is a *static analysis tool* (static checker) which uses a superset of the language, allowing you to add type annotations to all of your code and catch an entire class of bugs at compile time.

PropTypes is a *basic type checker* (runtime checker) which has been patched onto React. It can't check anything other than the types of the props being passed to a given component. If you want more flexible typechecking for your entire project Flow/TypeScript are appropriate choices.

143. How to use Font Awesome icons in React?

The below steps followed to include Font Awesome in React:

1. Install `font-awesome`:

```
$ npm install --save font-awesome
```

2. Import `font-awesome` in your `index.js` file:

```
import "font-awesome/css/font-awesome.min.css";
```

3. Add Font Awesome classes in `className`:

```
function MyComponent {
  return <div><i className={'fa fa-spinner'} /></div>
}
```

144. What is React Dev Tools?

React Developer Tools let you inspect the component hierarchy, including component props and state. It exists both as a browser extension (for Chrome and Firefox), and as a standalone app (works with other environments including Safari, IE, and React Native).

The official extensions available for different browsers or environments.

1. **Chrome extension**
2. **Firefox extension**
3. **Standalone app** (Safari, React Native, etc)

145. Why is DevTools not loading in Chrome for local files?

If you opened a local HTML file in your browser (`file://...`) then you must first open *Chrome Extensions* and check `Allow access to file URLs`.

146. How to use Polymer in React?

You need to follow below steps to use Polymer in React,

1. Create a Polymer element:

```
<link
  rel="import"
  href="../../bower_components/polymer/polymer.html"
/>;
Polymer({
  is: "calendar-element",
  ready: function () {
    this.textContent = "I am a calendar";
  },
});
```

2. Create the Polymer component HTML tag by importing it in a HTML document, e.g. import it in the `index.html` of your React application:

```
<link
  rel="import"
  href="./src/polymer-components/calendar-element.html"
/>
```

3. Use that element in the JSX file:

```
export default function MyComponent {
  return <calendar-element />;
}
```

147. What are the advantages of React over Vue.js?

React has the following advantages over Vue.js:

1. Gives more flexibility in large apps developing.
2. Easier to test.
3. Suitable for mobile apps creating.
4. More information and solutions available.

Note: The above list of advantages are purely opinionated and it vary based on the professional experience. But they are helpful as base parameters.

148. What is the difference between React and Angular?

Let's see the difference between React and Angular in a table format.

React	**Angular**
React is a library and has only the View layer	Angular is a framework and has complete MVC functionality
React handles rendering on the server side	AngularJS renders only on the client side but Angular 2 and above renders on the server side
React uses JSX that looks like HTML in JS which can be confusing	Angular follows the template approach for HTML, which makes code shorter and easy to understand
React Native, which is a React type to build mobile applications are faster and more stable	Ionic, Angular's mobile native app is relatively less stable and slower
In React, data flows only in one way and hence debugging is easy	In Angular, data flows both way i.e it has two-way data binding between children and parent and hence debugging is often difficult

Note: The above list of differences are purely opinionated and it vary based on the professional experience. But they are helpful as base parameters.

149. Why React tab is not showing up in DevTools?

When the page loads, *React DevTools* sets a global named `__REACT_DEVTOOLS_GLOBAL_HOOK__`, then React communicates with that hook during initialization. If the website is not using React or if React fails to communicate with DevTools then it won't show up the tab.

150. What are Styled Components?

`styled-components` is a JavaScript library for styling React applications. It removes the mapping between styles and components, and lets you write actual CSS augmented with JavaScript.

151. Give an example of Styled Components?

Lets create `<Title>` and `<Wrapper>` components with specific styles for each.

```
import React from "react";
import styled from "styled-components";

// Create a <Title> component that renders an <h1> which is centered, red and sized
```

```
at 1.5em
const Title = styled.h1`
  font-size: 1.5em;
  text-align: center;
  color: palevioletred;
`;

// Create a <Wrapper> component that renders a <section> with some padding and a
papayawhip background
const Wrapper = styled.section`
  padding: 4em;
  background: papayawhip;
`;
```

These two variables, `Title` and `Wrapper`, are now components that you can render just like any other react component.

```
<Wrapper>
  <Title>{"Lets start first styled component!"}</Title>
</Wrapper>
```

152. What is Relay?

Relay is a JavaScript framework for providing a data layer and client-server communication to web applications using the React view layer.

Miscellaneous

153. What are the main features of Reselect library?

Let's see the main features of Reselect library,

1. Selectors can compute derived data, allowing Redux to store the minimal possible state.
2. Selectors are efficient. A selector is not recomputed unless one of its arguments changes.
3. Selectors are composable. They can be used as input to other selectors.

154. **Give an example of Reselect usage?**

Let's take calculations and different amounts of a shipment order with the simplified usage of Reselect:

```
import { createSelector } from "reselect";

const shopItemsSelector = (state) => state.shop.items;
const taxPercentSelector = (state) => state.shop.taxPercent;

const subtotalSelector = createSelector(shopItemsSelector, (items) =>
  items.reduce((acc, item) => acc + item.value, 0)
);

const taxSelector = createSelector(
  subtotalSelector,
  taxPercentSelector,
```

```
  (subtotal, taxPercent) => subtotal * (taxPercent / 100)
);

export const totalSelector = createSelector(
  subtotalSelector,
  taxSelector,
  (subtotal, tax) => ({ total: subtotal + tax })
);

let exampleState = {
  shop: {
    taxPercent: 8,
    items: [
      { name: "apple", value: 1.2 },
      { name: "orange", value: 0.95 },
    ],
  },
};

console.log(subtotalSelector(exampleState)); // 2.15
console.log(taxSelector(exampleState)); // 0.172
console.log(totalSelector(exampleState)); // { total: 2.322 }
```

155. Can Redux only be used with React?

Redux can be used as a data store for any UI layer. The most common usage is with React and React Native, but there are bindings available for Angular, Angular 2, Vue, Mithril, and more. Redux simply provides a subscription mechanism which can be used by any other code.

156. Do you need to have a particular build tool to use Redux?

Redux is originally written in ES6 and transpiled for production into ES5 with Webpack and Babel. You should be able to use it regardless of your JavaScript build process. Redux also offers a UMD build that can be used directly without any build process at all.

157. How Redux Form `initialValues` get updated from state?

You need to add `enableReinitialize : true` setting.

```
const InitializeFromStateForm = reduxForm({
  form: "initializeFromState",
  enableReinitialize: true,
})(UserEdit);
```

If your `initialValues` prop gets updated, your form will update too.

158. How React PropTypes allow different types for one prop?

You can use `oneOfType()` method of `PropTypes`.

For example, the height property can be defined with either `string` or `number` type as below:

```
Component.propTypes = {
  size: PropTypes.oneOfType([PropTypes.string, PropTypes.number]),
};
```

159. Can I import an SVG file as react component?

You can import SVG directly as component instead of loading it as a file. This feature is available with `react-scripts@2.0.0` and higher.

```
import { ReactComponent as Logo } from "./logo.svg";

const App = () => (
  <div>
    {/* Logo is an actual react component */}
    <Logo />
  </div>
);
```

Note: Don't forget about the curly braces in the import.

160. What is render hijacking in react?

The concept of render hijacking is the ability to control what a component will output from another component. It means that you decorate your component by wrapping it into a Higher-Order component. By wrapping, you can inject additional props or make other changes, which can cause changing logic of rendering. It does not actually enable hijacking, but by using HOC you make your component behave differently.

161. How to pass numbers to React component?

We can pass `numbers` as `props` to React component using curly braces `{}` where as `strings` in double quotes `""` or single quotes `''`

```
import React from "react";

const ChildComponent = ({ name, age }) => {
  return (
    <>
      My Name is {name} and Age is {age}
    </>
  );
};

const ParentComponent = () => {
  return (
    <>
      <ChildComponent name="Chetan" age={30} />
    </>
  );
};

export default ParentComponent;
```

162. Do I need to keep all my state into Redux? Should I ever use react internal state?

It is up to the developer's decision, i.e., it is developer's job to determine what kinds of state make up your application, and where each piece of state should live. Some users prefer to keep every single piece of data in Redux, to maintain a fully serializable and controlled version of their application at all times. Others prefer to keep non-critical or UI state, such as "is this dropdown currently open", inside a component's internal state.

Below are the rules of thumb to determine what kind of data should be put into Redux

1. Do other parts of the application care about this data?
2. Do you need to be able to create further derived data based on this original data?
3. Is the same data being used to drive multiple components?
4. Is there value to you in being able to restore this state to a given point in time (ie, time travel debugging)?
5. Do you want to cache the data (i.e, use what's in state if it's already there instead of re-requesting it)?

163. What is the purpose of registerServiceWorker in React?

React creates a service worker for you without any configuration by default. The service worker is a web API that helps you cache your assets and other files so that when the user is offline or on a slow network, he/she can still see results on the screen, as such, it helps you build a better user experience, that's what you should know about service worker for now. It's all about adding offline capabilities to your site.

```
import React from "react";
import ReactDOM from "react-dom";
import App from "./App";
import registerServiceWorker from "./registerServiceWorker";

ReactDOM.render(<App />, document.getElementById("root"));
registerServiceWorker();
```

164. What is React memo function?

Class components can be restricted from re-rendering when their input props are the same using **PureComponent or shouldComponentUpdate**. Now you can do the same with function components by wrapping them in **React.memo**.

```
const MyComponent = React.memo(function MyComponent(props) {
  /* only rerenders if props change */
});
```

165. What is React lazy function?

The `React.lazy` function lets you render a dynamic import as a regular component. It will automatically load the bundle containing the `OtherComponent` when the component gets rendered. This must return a Promise which resolves to a module with a default export containing a React component.

```
const OtherComponent = React.lazy(() => import("./OtherComponent"));
```

```
function MyComponent() {
  return (
    <div>
      <OtherComponent />
    </div>
  );
}
```

Note:
`React.lazy` and `Suspense` is not yet available for server-side rendering. If you want to do code-splitting in a server rendered app, we still recommend React Loadable.

166. How to prevent unnecessary updates using setState?

You can compare the current value of the state with an existing state value and decide whether to rerender the page or not. If the values are the same then you need to return **null** to stop re-rendering otherwise return the latest state value.

For example, the user profile information is conditionally rendered as follows,

```
getUserProfile = (user) => {
  const latestAddress = user.address;
  this.setState((state) => {
    if (state.address === latestAddress) {
      return null;
    } else {
      return { title: latestAddress };
    }
  });
};
```

167. How do you render Array, Strings and Numbers in React 16 Version?

Arrays: Unlike older releases, you don't need to make sure **render** method return a single element in React16. You are able to return multiple sibling elements without a wrapping element by returning an array.

For example, let us take the below list of developers,

```
const ReactJSDevs = () => {
  return [
    <li key="1">John</li>,
    <li key="2">Jackie</li>,
    <li key="3">Jordan</li>,
  ];
};
```

You can also merge this array of items in another array component.

```
const JSDevs = () => {
  return (
```

```
    <ul>
      <li>Brad</li>
      <li>Brodge</li>
      <ReactJSDevs />
      <li>Brandon</li>
    </ul>
  );
};
```

Strings and Numbers: You can also return string and number type from the render method.

```
render() {
 return 'Welcome to ReactJS questions';
}
// Number
render() {
 return 2018;
}
```

168. What are hooks?

Hooks is a special JavaScript function that allows you use state and other React features without writing a class. This pattern has been introduced as a new feature in React 16.8 and helped to isolate the stateful logic from the components.

Let's see an example of useState hook:

```
import { useState } from "react";

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </>
  );
}
```

Note: Hooks can be used inside an existing function component without rewriting the component.

169. What rules need to be followed for hooks?

You need to follow two rules in order to use hooks,

1. **Call Hooks only at the top level of your react functions:** You should always use hooks at the top level of react function before any early returns. i.e, You shouldn't call Hooks inside loops, conditions, or nested functions. This will ensure that Hooks are called in the same order each time a component renders and it preserves the state of Hooks between multiple re-renders due to useState and useEffect calls.

2. **Call Hooks from React Functions only:** You shouldn't call Hooks from regular JavaScript functions or class components. Instead, you should call them from either function components or custom hooks.

170. How to ensure hooks followed the rules in your project?

React team released an ESLint plugin called **eslint-plugin-react-hooks** that enforces Hook's two rules. It is part of Hooks API. You can add this plugin to your project using the below command,

```
npm install eslint-plugin-react-hooks --save-dev
```

And apply the below config in your ESLint config file,

```
// Your ESLint configuration
{
  "plugins": [
    // ...
    "react-hooks"
  ],
  "rules": {
    // ...
    "react-hooks/rules-of-hooks": "error"
  }
}
```

The recommended `eslint-config-react-app` preset already includes the hooks rules of this plugin.
For example, the linter enforce proper naming convention for hooks. If you rename your custom hooks which as prefix "use" to something else then linter won't allow you to call built-in hooks such as useState, useEffect etc inside of your custom hook anymore.

Note: This plugin is intended to use in Create React App by default.

171. What are the differences between Flux and Redux?

Below are the major differences between Flux and Redux

Flux	Redux
State is mutable	State is immutable
The Store contains both state and change logic	The Store and change logic are separate
There are multiple stores exist	There is only one store exist
All the stores are disconnected and flat	Single store with hierarchical reducers
It has a singleton dispatcher	There is no concept of dispatcher
React components subscribe to the store	Container components uses connect function

172. What are the benefits of React Router V4?

Below are the main benefits of React Router V4 module,

1. In React Router v4(version 4), the API is completely about components. A router can be visualized as a single component(`<BrowserRouter>`) which wraps specific child router components(`<Route>`).
2. You don't need to manually set history. The router module will take care history by wrapping routes with `<BrowserRouter>` component.
3. The application size is reduced by adding only the specific router module(Web, core, or native)

173. Can you describe about componentDidCatch lifecycle method signature?

The **componentDidCatch** lifecycle method is invoked after an error has been thrown by a descendant component. The method receives two parameters,

1. error: - The error object which was thrown
2. info: - An object with a componentStack key contains the information about which component threw the error.

The method structure would be as follows

```
componentDidCatch(error, info);
```

174. In which scenarios do error boundaries not catch errors?

Below are the cases in which error boundaries don't work,

1. Inside Event handlers
2. Asynchronous code using **setTimeout or requestAnimationFrame** callbacks
3. During Server side rendering
4. When errors thrown in the error boundary code itself

175. What is the behavior of uncaught errors in react 16?

In React 16, errors that were not caught by any error boundary will result in unmounting of the whole React component tree. The reason behind this decision is that it is worse to leave corrupted UI in place than to completely remove it. For example, it is worse for a payments app to display a wrong amount than to render nothing.

176. What is the proper placement for error boundaries?

The granularity of error boundaries usage is up to the developer based on project needs. You can follow either of these approaches,

1. You can wrap top-level route components to display a generic error message for the entire application.
2. You can also wrap individual components in an error boundary to protect them from crashing the rest of the application.

177. What is the benefit of component stack trace from error boundary?

Apart from error messages and javascript stack, React16 will display the component stack trace with file names and line numbers using error boundary concept.

For example, BuggyCounter component displays the component stack trace as below,

```
▶React caught an error thrown by BuggyCounter. You should fix this error in your code.    react-dom.development.js:7708
React will try to recreate this component tree from scratch using the error boundary you provided, ErrorBoundary.

Error: I crashed!

The error is located at:
    in BuggyCounter (at App.js:26)
    in ErrorBoundary (at App.js:21)
    in div (at App.js:8)
    in App (at index.js:5)
```

178. What are default props?

The *defaultProps* can be defined as a property on the component to set the default values for the props. These default props are used when props not supplied(i.e., undefined props), but not for `null` or `0` as props. That means, If you provide null value then it remains null value. It's the same behavior with 0 as well.

For example, let us create color default prop for the button component,

```
function MyButton {
  // ...
}

MyButton.defaultProps = {
  color: "red",
};
```

If `props.color` is not provided then it will set the default value to 'red'. i.e, Whenever you try to access the color prop it uses the default value

```
function MyButton() {
  return <MyButton />; // props.color will contain red value
}
```

179. What is the purpose of displayName class property?

The displayName string is used in debugging messages. Usually, you don't need to set it explicitly because it's inferred from the name of the function or class that defines the component. You might want to set it explicitly if you want to display a different name for debugging purposes or when you create a higher-order component.

For example, To ease debugging, choose a display name that communicates that it's the result of a withSubscription HOC.

```
function withSubscription(WrappedComponent) {
  class WithSubscription extends React.Component {
    /* ... */
  }
  WithSubscription.displayName = `WithSubscription(${getDisplayName(
    WrappedComponent
  )})`;
  return WithSubscription;
}
function getDisplayName(WrappedComponent) {
  return (
    WrappedComponent.displayName || WrappedComponent.name || "Component"
```

```
    );
  }
```

180. What is the browser support for react applications?

React supports all popular browsers, including Internet Explorer 9 and above, although some polyfills are required for older browsers such as IE 9 and IE 10. If you use **es5-shim and es5-sham** polyfill then it even support old browsers that doesn't support ES5 methods.

181. What is code-splitting?

Code-Splitting is a feature supported by bundlers like Webpack and Browserify which can create multiple bundles that can be dynamically loaded at runtime. The react project supports code splitting via dynamic import() feature.

For example, in the below code snippets, it will make moduleA.js and all its unique dependencies as a separate chunk that only loads after the user clicks the 'Load' button.

moduleA.js

```
const moduleA = "Hello";

export { moduleA };
```

App.js

```
export default function App {
  function handleClick() {
    import("./moduleA")
      .then(({ moduleA }) => {
        // Use moduleA
      })
      .catch((err) => {
        // Handle failure
      });
  };

 return (
   <div>
     <button onClick={this.handleClick}>Load</button>
   </div>
 );
}
```

▶ **See Class**

```
import React, { Component } from "react";

class App extends Component {
  handleClick = () => {
```

```
      import("./moduleA")
        .then(({ moduleA }) => {
          // Use moduleA
        })
        .catch((err) => {
          // Handle failure
        });
    };

    render() {
      return (
        <div>
          <button onClick={this.handleClick}>Load</button>
        </div>
      );
    }
  }

  export default App;
```

182. What are Keyed Fragments?

The Fragments declared with the explicit <React.Fragment> syntax may have keys. The general use case is mapping a collection to an array of fragments as below,

```
function Glossary(props) {
  return (
    <dl>
      {props.items.map((item) => (
        // Without the `key`, React will fire a key warning
        <React.Fragment key={item.id}>
          <dt>{item.term}</dt>
          <dd>{item.description}</dd>
        </React.Fragment>
      ))}
    </dl>
  );
}
```

Note: key is the only attribute that can be passed to Fragment. In the future, there might be a support for additional attributes, such as event handlers.

183. Does React support all HTML attributes?

As of React 16, both standard or custom DOM attributes are fully supported. Since React components often take both custom and DOM-related props, React uses the camelCase convention just like the DOM APIs.

Let us take few props with respect to standard HTML attributes,

```
<div tabIndex="-1" />      // Just like node.tabIndex DOM API
<div className="Button" /> // Just like node.className DOM API
<input readOnly={true} />  // Just like node.readOnly DOM API
```

These props work similarly to the corresponding HTML attributes, with the exception of the special cases. It also support all SVG attributes.

184. ### When component props defaults to true?

If you pass no value for a prop, it defaults to true. This behavior is available so that it matches the behavior of HTML.

For example, below expressions are equivalent,

```
<MyInput autocomplete />

<MyInput autocomplete={true} />
```

Note: It is not recommended to use this approach because it can be confused with the ES6 object shorthand (example, `{name}` which is short for `{name: name}`)

185. ### What is NextJS and major features of it?

Next.js is a popular and lightweight framework for static and server-rendered applications built with React. It also provides styling and routing solutions. Below are the major features provided by NextJS,

1. Server-rendered by default
2. Automatic code splitting for faster page loads
3. Simple client-side routing (page based)
4. Webpack-based dev environment which supports (HMR)
5. Able to implement with Express or any other Node.js HTTP server
6. Customizable with your own Babel and Webpack configurations

186. ### How do you pass an event handler to a component?

You can pass event handlers and other functions as props to child components. The functions can be passed to child component as below,

```
function Button({ onClick }) {
  return <button onClick={onClick}>Download</button>;
}

export default function downloadExcel() {
  function handleClick() {
    alert("Downloaded");
  }

  return <Button onClick={handleClick}></Button>;
}
```

187. ### How to prevent a function from being called multiple times?

If you use an event handler such as **onClick or onScroll** and want to prevent the callback from being fired too quickly, then you can limit the rate at which callback is executed. This can be achieved in the below possible

ways,

1. **Throttling:** Changes based on a time based frequency. For example, it can be used using _.throttle lodash function
2. **Debouncing:** Publish changes after a period of inactivity. For example, it can be used using _.debounce lodash function
3. **RequestAnimationFrame throttling:** Changes based on requestAnimationFrame. For example, it can be used using raf-schd lodash function

188. How JSX prevents Injection Attacks?

React DOM escapes any values embedded in JSX before rendering them. Thus it ensures that you can never inject anything that's not explicitly written in your application. Everything is converted to a string before being rendered.

For example, you can embed user input as below,

```
const name = response.potentiallyMaliciousInput;
const element = <h1>{name}</h1>;
```

This way you can prevent XSS(Cross-site-scripting) attacks in the application.

189. How do you update rendered elements?

You can update UI(represented by rendered element) by passing the newly created element to ReactDOM's render method.

For example, lets take a ticking clock example, where it updates the time by calling render method multiple times,

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(element, document.getElementById("root"));
}

setInterval(tick, 1000);
```

190. How do you say that props are readonly?

When you declare a component as a function or a class, it must never modify its own props.

Let us take a below capital function,

```
function capital(amount, interest) {
  return amount + interest;
}
```

The above function is called “pure” because it does not attempt to change their inputs, and always return the same result for the same inputs. Hence, React has a single rule saying "All React components must act like pure functions with respect to their props."

191. What are the conditions to safely use the index as a key?

There are three conditions to make sure, it is safe use the index as a key.

1. The list and items are static– they are not computed and do not change
2. The items in the list have no ids
3. The list is never reordered or filtered.

192. Should keys be globally unique?

The keys used within arrays should be unique among their siblings but they don’t need to be globally unique. i.e, You can use the same keys with two different arrays.

For example, the below `Book` component uses two arrays with different arrays,

```
function Book(props) {
  const index = (
    <ul>
      {props.pages.map((page) => (
        <li key={page.id}>{page.title}</li>
      ))}
    </ul>
  );
  const content = props.pages.map((page) => (
    <div key={page.id}>
      <h3>{page.title}</h3>
      <p>{page.content}</p>
      <p>{page.pageNumber}</p>
    </div>
  ));
  return (
    <div>
      {index}
      <hr />
      {content}
    </div>
  );
}
```

193. What is the popular choice for form handling?

`Formik` is a form library for react which provides solutions such as validation, keeping track of the visited fields, and handling form submission.

In detail, You can categorize them as follows,

1. Getting values in and out of form state
2. Validation and error messages

3. Handling form submission

It is used to create a scalable, performant, form helper with a minimal API to solve annoying stuff.

194. What are the advantages of formik over redux form library?

Below are the main reasons to recommend formik over redux form library,

1. The form state is inherently short-term and local, so tracking it in Redux (or any kind of Flux library) is unnecessary.
2. Redux-Form calls your entire top-level Redux reducer multiple times ON EVERY SINGLE KEYSTROKE. This way it increases input latency for large apps.
3. Redux-Form is 22.5 kB minified gzipped whereas Formik is 12.7 kB

195. Why are you not required to use inheritance?

In React, it is recommended to use composition over inheritance to reuse code between components. Both Props and composition give you all the flexibility you need to customize a component's look and behavior explicitly and safely.
Whereas, If you want to reuse non-UI functionality between components, it is suggested to extract it into a separate JavaScript module. Later components import it and use that function, object, or class, without extending it.

196. Can I use web components in react application?

Yes, you can use web components in a react application. Even though many developers won't use this combination, it may require especially if you are using third-party UI components that are written using Web Components.

For example, let us use `Vaadin` date picker web component as below,

```
import "./App.css";
import "@vaadin/vaadin-date-picker";
export default function App() {
  return (
    <div className="App">
      <vaadin-date-picker label="When were you born?"></vaadin-date-picker>
    </div>
  );
}
```

197. What is dynamic import?

You can achieve code-splitting in your app using dynamic import.

Let's take an example of addition,

1. **Normal Import**

```
import { add } from "./math";
console.log(add(10, 20));
```

2. **Dynamic Import**

```
import("./math").then((math) => {
  console.log(math.add(10, 20));
});
```

198. What are loadable components?

With the release of React 18, React.lazy and Suspense are now available for server-side rendering. However, prior to React 18, it was recommended to use Loadable Components for code-splitting in a server-side rendered app because React.lazy and Suspense were not available for server-side rendering. Loadable Components lets you render a dynamic import as a regular component. For example, you can use Loadable Components to load the OtherComponent in a separate bundle like this:

```
import loadable from "@loadable/component";

const OtherComponent = loadable(() => import("./OtherComponent"));

function MyComponent() {
  return (
    <div>
      <OtherComponent />
    </div>
  );
}
```

Now OtherComponent will be loaded in a separated bundle
Loadable Components provides additional benefits beyond just code-splitting, such as automatic code reloading, error handling, and preloading. By using Loadable Components, you can ensure that your application loads quickly and efficiently, providing a better user experience for your users.

199. What is suspense component?

If the module containing the dynamic import is not yet loaded by the time parent component renders, you must show some fallback content while you're waiting for it to load using a loading indicator. This can be done using **Suspense** component.

For example, the below code uses suspense component,

```
const OtherComponent = React.lazy(() => import("./OtherComponent"));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <OtherComponent />
      </Suspense>
    </div>
  );
}
```

As mentioned in the above code, Suspense is wrapped above the lazy component.

200. What is route based code splitting?

One of the best place to do code splitting is with routes. The entire page is going to re-render at once so users are unlikely to interact with other elements in the page at the same time. Due to this, the user experience won't be disturbed.

Let us take an example of route based website using libraries like React Router with React.lazy,

```
import { BrowserRouter as Router, Route, Switch } from "react-router-dom";
import React, { Suspense, lazy } from "react";

const Home = lazy(() => import("./routes/Home"));
const About = lazy(() => import("./routes/About"));

const App = () => (
  <Router>
    <Suspense fallback={<div>Loading...</div>}>
      <Switch>
        <Route exact path="/" component={Home} />
        <Route path="/about" component={About} />
      </Switch>
    </Suspense>
  </Router>
);
```

In the above code, the code splitting will happen at each route level.

201. What is the purpose of default value in context?

The defaultValue argument is only used when a component does not have a matching Provider above it in the tree. This can be helpful for testing components in isolation without wrapping them.

Below code snippet provides default theme value as Luna.

```
const MyContext = React.createContext(defaultValue);
```

202. What is diffing algorithm?

React needs to use algorithms to find out how to efficiently update the UI to match the most recent tree. The diffing algorithms is generating the minimum number of operations to transform one tree into another. However, the algorithms have a complexity in the order of $O(n^3)$ where n is the number of elements in the tree.

In this case, displaying 1000 elements would require in the order of one billion comparisons. This is far too expensive. Instead, React implements a heuristic O(n) algorithm based on two assumptions:

1. Two elements of different types will produce different trees.
2. The developer can hint at which child elements may be stable across different renders with a key prop.

203. What are the rules covered by diffing algorithm?

When diffing two trees, React first compares the two root elements. The behavior is different depending on the types of the root elements. It covers the below rules during reconciliation algorithm,

1. **Elements Of Different Types:**
   Whenever the root elements have different types, React will tear down the old tree and build the new tree from scratch. For example, elements to , or from

   to of different types lead a full rebuild.

2. **DOM Elements Of The Same Type:**
   When comparing two React DOM elements of the same type, React looks at the attributes of both, keeps the same underlying DOM node, and only updates the changed attributes. Lets take an example with same DOM elements except className attribute,

   ```
   <div className="show" title="ReactJS" />

   <div className="hide" title="ReactJS" />
   ```

3. **Component Elements Of The Same Type:**
   When a component updates, the instance stays the same, so that state is maintained across renders. React updates the props of the underlying component instance to match the new element, and calls componentWillReceiveProps() and componentWillUpdate() on the underlying instance. After that, the render() method is called and the diff algorithm recurses on the previous result and the new result.

4. **Recursing On Children:**
   when recursing on the children of a DOM node, React just iterates over both lists of children at the same time and generates a mutation whenever there's a difference. For example, when adding an element at the end of the children, converting between these two trees works well.

   ```
   <ul>
     <li>first</li>
     <li>second</li>
   </ul>

   <ul>
     <li>first</li>
     <li>second</li>
     <li>third</li>
   </ul>
   ```

5. **Handling keys:**
   React supports a key attribute. When children have keys, React uses the key to match children in the original tree with children in the subsequent tree. For example, adding a key can make the tree conversion efficient,

```
<ul>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>
```

```
<ul>
  <li key="2014">Connecticut</li>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>
```

204. When do you need to use refs?

There are few use cases to go for refs,

1. Managing focus, text selection, or media playback.
2. Triggering imperative animations.
3. Integrating with third-party DOM libraries.

205. Must prop be named as render for render props?

Even though the pattern named render props, you don't have to use a prop named render to use this pattern. i.e, Any prop that is a function that a component uses to know what to render is technically a "render prop". Lets take an example with the children prop for render props,

```
<Mouse
  children={(mouse) => (
    <p>
      The mouse position is {mouse.x}, {mouse.y}
    </p>
  )}
/>
```

Actually children prop doesn't need to be named in the list of "attributes" in JSX element. Instead, you can keep it directly inside element,

```
<Mouse>
  {(mouse) => (
    <p>
      The mouse position is {mouse.x}, {mouse.y}
    </p>
  )}
</Mouse>
```

While using this above technique(without any name), explicitly state that children should be a function in your propTypes.

```
Mouse.propTypes = {
  children: PropTypes.func.isRequired,
};
```

206. What are the problems of using render props with pure components?

If you create a function inside a render method, it negates the purpose of pure component. Because the shallow prop comparison will always return false for new props, and each render in this case will generate a new value for the render prop. You can solve this issue by defining the render function as instance method.

207. What is windowing technique?

Windowing is a technique that only renders a small subset of your rows at any given time, and can dramatically reduce the time it takes to re-render the components as well as the number of DOM nodes created. If your application renders long lists of data then this technique is recommended. Both react-window and react-virtualized are popular windowing libraries which provides several reusable components for displaying lists, grids, and tabular data.

208. How do you print falsy values in JSX?

The falsy values such as false, null, undefined, and true are valid children but they don't render anything. If you still want to display them then you need to convert it to string. Let's take an example on how to convert to a string,

```
<div>My JavaScript variable is {String(myVariable)}.</div>
```

209. What is the typical use case of portals?

React portals are very useful when a parent component has overflow: hidden or has properties that affect the stacking context (e.g. z-index, position, opacity) and you need to visually “break out” of its container.

For example, dialogs, global message notifications, hovercards, and tooltips.

210. How do you set default value for uncontrolled component?

In React, the value attribute on form elements will override the value in the DOM. With an uncontrolled component, you might want React to specify the initial value, but leave subsequent updates uncontrolled. To handle this case, you can specify a **defaultValue** attribute instead of **value**.

```
render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <label>
        User Name:
        <input
          defaultValue="John"
          type="text"
          ref={this.input} />
      </label>
      <input type="submit" value="Submit" />
    </form>
  );
}
```

The same applies for `select` and `textArea` inputs. But you need to use **defaultChecked** for `checkbox` and `radio` inputs.

211. What is your favorite React stack?

Even though the tech stack varies from developer to developer, the most popular stack is used in react boilerplate project code. It mainly uses Redux and redux-saga for state management and asynchronous side-effects, react-router for routing purpose, styled-components for styling react components, axios for invoking REST api, and other supported stack such as webpack, reselect, ESNext, Babel.
You can clone the project https://github.com/react-boilerplate/react-boilerplate and start working on any new react project.

212. What is the difference between Real DOM and Virtual DOM?

Below are the main differences between Real DOM and Virtual DOM,

Real DOM	**Virtual DOM**
Updates are slow	Updates are fast
DOM manipulation is very expensive.	DOM manipulation is very easy
You can update HTML directly.	You Can't directly update HTML
It causes too much of memory wastage	There is no memory wastage
Creates a new DOM if element updates	It updates the JSX if element update

213. How to add Bootstrap to a react application?

Bootstrap can be added to your React app in a three possible ways,

1. Using the Bootstrap CDN:
   This is the easiest way to add bootstrap. Add both bootstrap CSS and JS resources in a head tag.
2. Bootstrap as Dependency:
   If you are using a build tool or a module bundler such as Webpack, then this is the preferred option for adding Bootstrap to your React application

   ```
   npm install bootstrap
   ```

3. React Bootstrap Package:
   In this case, you can add Bootstrap to our React app is by using a package that has rebuilt Bootstrap components to work particularly as React components. Below packages are popular in this category,
   1. react-bootstrap
   2. reactstrap

214. Can you list down top websites or applications using react as front end framework?

Below are the `top 10 websites` using React as their front-end framework,

1. Facebook
2. Uber
3. Instagram
4. WhatsApp
5. Khan Academy
6. Airbnb

7. Dropbox
8. Flipboard
9. Netflix
10. PayPal

215. Is it recommended to use CSS In JS technique in React?

React does not have any opinion about how styles are defined but if you are a beginner then good starting point is to define your styles in a separate *.css file as usual and refer to them using className. This functionality is not part of React but came from third-party libraries. But If you want to try a different approach(CSS-In-JS) then styled-components library is a good option.

216. Do I need to rewrite all my class components with hooks?

No. But you can try Hooks in a few components(or new components) without rewriting any existing code. Because there are no plans to remove classes in ReactJS.

217. How to fetch data with React Hooks?

The effect hook called `useEffect` can be used to fetch data from an API and to set the data in the local state of the component with the useState hook's update function.

Here is an example of fetching a list of react articles from an API using fetch.

```
import React from "react";

function App() {
  const [data, setData] = React.useState({ hits: [] });

  React.useEffect(() => {
    fetch("http://hn.algolia.com/api/v1/search?query=react")
      .then((response) => response.json())
      .then((data) => setData(data));
  }, []);

  return (
    <ul>
      {data.hits.map((item) => (
        <li key={item.objectID}>
          <a href={item.url}>{item.title}</a>
        </li>
      ))}
    </ul>
  );
}

export default App;
```

A popular way to simplify this is by using the library axios.

We provided an empty array as second argument to the useEffect hook to avoid activating it on component updates. This way, it only fetches on component mount.

218. Is Hooks cover all use cases for classes?

Hooks doesn't cover all use cases of classes but there is a plan to add them soon. Currently there are no Hook equivalents to the uncommon **getSnapshotBeforeUpdate** and **componentDidCatch** lifecycles yet.

219. What is the stable release for hooks support?

React includes a stable implementation of React Hooks in 16.8 release for below packages

1. React DOM
2. React DOM Server
3. React Test Renderer
4. React Shallow Renderer

220. Why do we use array destructuring (square brackets notation) in `useState`?

When we declare a state variable with `useState`, it returns a pair — an array with two items. The first item is the current value, and the second is a function that updates the value. Using [0] and [1] to access them is a bit confusing because they have a specific meaning. This is why we use array destructuring instead.

For example, the array index access would look as follows:

```
var userStateVariable = useState("userProfile"); // Returns an array pair
var user = userStateVariable[0]; // Access first item
var setUser = userStateVariable[1]; // Access second item
```

Whereas with array destructuring the variables can be accessed as follows:

```
const [user, setUser] = useState("userProfile");
```

221. What are the sources used for introducing hooks?

Hooks got the ideas from several different sources. Below are some of them,

1. Previous experiments with functional APIs in the react-future repository
2. Community experiments with render prop APIs such as Reactions Component
3. State variables and state cells in DisplayScript.
4. Subscriptions in Rx.
5. Reducer components in ReasonReact.

222. How do you access imperative API of web components?

Web Components often expose an imperative API to implement its functions. You will need to use a **ref** to interact with the DOM node directly if you want to access imperative API of a web component. But if you are using third-party Web Components, the best solution is to write a React component that behaves as a **wrapper** for your Web Component.

223. What is formik?

Formik is a small react form library that helps you with the three major problems,

1. Getting values in and out of form state
2. Validation and error messages

3. Handling form submission

224. What are typical middleware choices for handling asynchronous calls in Redux?

Some of the popular middleware choices for handling asynchronous calls in Redux eco system are `Redux Thunk, Redux Promise, Redux Saga`.

225. Do browsers understand JSX code?

No, browsers can't understand JSX code. You need a transpiler to convert your JSX to regular Javascript that browsers can understand. The most widely used transpiler right now is Babel.

226. Describe about data flow in react?

React implements one-way reactive data flow using props which reduce boilerplate and is easier to understand than traditional two-way data binding.

227. What is MobX?

MobX is a simple, scalable and battle tested state management solution for applying functional reactive programming (TFRP). For ReactJS application, you need to install below packages,

```
npm install mobx --save
npm install mobx-react --save
```

228. What are the differences between Redux and MobX?

Below are the main differences between Redux and MobX,

Topic	Redux	MobX
Definition	It is a javascript library for managing the application state	It is a library for reactively managing the state of your applications
Programming	It is mainly written in ES6	It is written in JavaScript(ES5)
Data Store	There is only one large store exist for data storage	There is more than one store for storage
Usage	Mainly used for large and complex applications	Used for simple applications
Performance	Need to be improved	Provides better performance
How it stores	Uses JS Object to store	Uses observable to store the data

229. Should I learn ES6 before learning ReactJS?

No, you don’t have to learn es2015/es6 to learn react. But you may find many resources or React ecosystem uses ES6 extensively. Let's see some of the frequently used ES6 features,

1. **Destructuring:** To get props and use them in a component

```
// in es 5
var someData = this.props.someData;
var dispatch = this.props.dispatch;

// in es6
const { someData, dispatch } = this.props;
```

2. **Spread operator:** Helps in passing props down into a component

```
// in es 5
<SomeComponent someData={this.props.someData} dispatch={this.props.dispatch}
/>

// in es6
<SomeComponent {...this.props} />
```

3. **Arrow functions:** Makes compact syntax

```
// es 5
var users = usersList.map(function (user) {
  return <li>{user.name}</li>;
});
// es 6
const users = usersList.map((user) => <li>{user.name}</li>);
```

230. What is Concurrent Rendering?

The Concurrent rendering makes React apps to be more responsive by rendering component trees without blocking the main UI thread. It allows React to interrupt a long-running render to handle a high-priority event. i.e, When you enabled concurrent Mode, React will keep an eye on other tasks that need to be done, and if there's something with a higher priority it will pause what it is currently rendering and let the other task finish first. You can enable this in two ways,

```
// 1. Part of an app by wrapping with ConcurrentMode
<React.unstable_ConcurrentMode>
  <Something />
</React.unstable_ConcurrentMode>;

// 2. Whole app using createRoot
ReactDOM.unstable_createRoot(domNode).render(<App />);
```

231. What is the difference between async mode and concurrent mode?

Both refers the same thing. Previously concurrent Mode being referred to as "Async Mode" by React team. The name has been changed to highlight React's ability to perform work on different priority levels. So it avoids the confusion from other approaches to Async Rendering.

232. Can I use javascript urls in react16.9?

Yes, you can use javascript: URLs but it will log a warning in the console. Because URLs starting with javascript: are dangerous by including unsanitized output in a tag like `<a href>` and create a security hole.

```
const companyProfile = {
  website: "javascript: alert('Your website is hacked')",
};
// It will log a warning
<a href={companyProfile.website}>More details</a>;
```

Remember that the future versions will throw an error for javascript URLs.

233. ## What is the purpose of eslint plugin for hooks?

The ESLint plugin enforces rules of Hooks to avoid bugs. It assumes that any function starting with “use” and a capital letter right after it is a Hook. In particular, the rule enforces that,

1. Calls to Hooks are either inside a PascalCase function (assumed to be a component) or another useSomething function (assumed to be a custom Hook).
2. Hooks are called in the same order on every render.

234. ## What is the difference between Imperative and Declarative in React?

Imagine a simple UI component, such as a "Like" button. When you tap it, it turns blue if it was previously grey, and grey if it was previously blue.

The imperative way of doing this would be:

```
if (user.likes()) {
  if (hasBlue()) {
    removeBlue();
    addGrey();
  } else {
    removeGrey();
    addBlue();
  }
}
```

Basically, you have to check what is currently on the screen and handle all the changes necessary to redraw it with the current state, including undoing the changes from the previous state. You can imagine how complex this could be in a real-world scenario.

In contrast, the declarative approach would be:

```
if (this.state.liked) {
  return <blueLike />;
} else {
  return <greyLike />;
}
```

Because the declarative approach separates concerns, this part of it only needs to handle how the UI should look in a specific state, and is therefore much simpler to understand.

235. What are the benefits of using TypeScript with ReactJS?

Below are some of the benefits of using TypeScript with ReactJS,

1. It is possible to use latest JavaScript features
2. Use of interfaces for complex type definitions
3. IDEs such as VS Code was made for TypeScript
4. Avoid bugs with the ease of readability and Validation

236. How do you make sure that user remains authenticated on page refresh while using Context API State Management?

When a user logs in and reload, to persist the state generally we add the load user action in the useEffect hooks in the main App.js. While using Redux, loadUser action can be easily accessed.

App.js

```
import { loadUser } from "../actions/auth";
store.dispatch(loadUser());
```

- But while using **Context API**, to access context in App.js, wrap the AuthState in index.js so that App.js can access the auth context. Now whenever the page reloads, no matter what route you are on, the user will be authenticated as **loadUser** action will be triggered on each re-render.

index.js

```
import React from "react";
import ReactDOM from "react-dom";
import App from "./App";
import AuthState from "./context/auth/AuthState";

ReactDOM.render(
  <React.StrictMode>
    <AuthState>
      <App />
    </AuthState>
  </React.StrictMode>,
  document.getElementById("root")
);
```

App.js

```
const authContext = useContext(AuthContext);

const { loadUser } = authContext;

useEffect(() => {
```

```
  loadUser();
}, []);
```

loadUser

```
const loadUser = async () => {
  const token = sessionStorage.getItem("token");

  if (!token) {
    dispatch({
      type: ERROR,
    });
  }
  setAuthToken(token);

  try {
    const res = await axios("/api/auth");

    dispatch({
      type: USER_LOADED,
      payload: res.data.data,
    });
  } catch (err) {
    console.error(err);
  }
};
```

237. What are the benefits of new JSX transform?

There are three major benefits of new JSX transform,

1. It is possible to use JSX without importing React packages
2. The compiled output might improve the bundle size in a small amount
3. The future improvements provides the flexibility to reduce the number of concepts to learn React.

238. How is the new JSX transform different from old transform??

The new JSX transform doesn't require React to be in scope. i.e, You don't need to import React package for simple scenarios.

Let's take an example to look at the main differences between the old and the new transform,

Old Transform:

```
import React from "react";

function App() {
  return <h1>Good morning!!</h1>;
}
```

Now JSX transform convert the above code into regular JavaScript as below,

```
import React from "react";

function App() {
  return React.createElement("h1", null, "Good morning!!");
}
```

New Transform:

The new JSX transform doesn't require any React imports

```
function App() {
  return <h1>Good morning!!</h1>;
}
```

Under the hood JSX transform compiles to below code

```
import { jsx as _jsx } from "react/jsx-runtime";

function App() {
  return _jsx("h1", { children: "Good morning!!" });
}
```

Note: You still need to import React to use Hooks.

239. What are React Server components?

React Server Component is a way to write React component that gets rendered in the server-side with the purpose of improving React app performance. These components allow us to load components from the backend.

Note: React Server Components is still under development and not recommended for production yet.

240. What is prop drilling?

Prop Drilling is the process by which you pass data from one component of the React Component tree to another by going through other components that do not need the data but only help in passing it around.

241. What is the difference between useState and useRef hook?

1. useState causes components to re-render after state updates whereas useRef doesn't cause a component to re-render when the value or state changes.
   Essentially, useRef is like a "box" that can hold a mutable value in its (.current) property.
2. useState allows us to update the state inside components. While useRef allows referencing DOM elements.

242. What is a wrapper component?

A wrapper in React is a component that wraps or surrounds another component or group of components. It can be used for a variety of purposes such as adding additional functionality, styling, or layout to the wrapped components.

For example, consider a simple component that displays a message:

```
const Message = ({ text }) => {
  return <p>{text}</p>;
};
```

We can create a wrapper component that will add a border to the message component:

```
const MessageWrapper = (props) => {
  return (
    <div style={{ border: "1px solid black" }}>
      <Message {...props} />
    </div>
  );
};
```

Now we can use the MessageWrapper component instead of the Message component and the message will be displayed with a border:

```
<MessageWrapper text="Hello World" />
```

Wrapper component can also accept its own props and pass them down to the wrapped component, for example, we can create a wrapper component that will add a title to the message component:

```
const MessageWrapperWithTitle = ({ title, ...props }) => {
  return (
    <div>
      <h3>{title}</h3>
      <Message {...props} />
    </div>
  );
};
```

Now we can use the MessageWrapperWithTitle component and pass title props:

```
<MessageWrapperWithTitle title="My Message" text="Hello World" />
```

This way, the wrapper component can add additional functionality, styling, or layout to the wrapped component while keeping the wrapped component simple and reusable.

243. What are the differences between useEffect and useLayoutEffect hooks?

useEffect and useLayoutEffect are both React hooks that can be used to synchronize a component with an external system, such as a browser API or a third-party library. However, there are some key differences between the two:

- Timing: useEffect runs after the browser has finished painting, while useLayoutEffect runs synchronously before the browser paints. This means that useLayoutEffect can be used to measure and update layout in a way that feels more synchronous to the user.
- Browser Paint: useEffect allows browser to paint the changes before running the effect, hence it may cause some visual flicker. useLayoutEffect synchronously runs the effect before browser paints and hence it will avoid visual flicker.
- Execution Order: The order in which multiple useEffect hooks are executed is determined by React and may not be predictable. However, the order in which multiple useLayoutEffect hooks are executed is determined by the order in which they were called.
- Error handling: useEffect has a built-in mechanism for handling errors that occur during the execution of the effect, so that it does not crash the entire application. useLayoutEffect does not have this mechanism, and errors that occur during the execution of the effect will crash the entire application.

In general, it's recommended to use useEffect as much as possible, because it is more performant and less prone to errors. useLayoutEffect should only be used when you need to measure or update layout, and you can't achieve the same result using useEffect.

244. What are the differences between Functional and Class Components?

There are two different ways to create components in ReactJS. The main differences are listed down as below,

1. Syntax:

The class components uses ES6 classes to create the components. It uses `render` function to display the HTML content in the webpage.

The syntax for class component looks like as below.

```
class App extends React.Component {
  render() {
    return <h1>This is a class component</h1>;
  }
}
```

Note: The **Pascal Case** is the recommended approach to provide naming to a component.

Functional component has been improved over the years with some added features like Hooks. Here is a syntax for functional component.

```
function App() {
  return (
    <div className="App">
      <h1>Hello, I'm a function component</h1>
    </div>
  );
}
```

2. State:

State contains information or data about a component which may change over time.

In class component, you can update the state when a user interacts with it or server updates the data using the `setState()` method. The initial state is going to be assigned in the `Constructor()` method using the `this.state` object and it is possible to assign different data types such as string, boolean, numbers, etc.

A simple example showing how we use the setState() and constructor():

```
class App extends Component {
  constructor() {
    super();
    this.state = {
      message: "This is a class component",
    };
  }
  updateMessage() {
    this.setState({
      message: "Updating the class component",
    });
  }
  render() {
    return (
      <>
        <h1>{this.state.message}</h1>
        <button
          onClick={() => {
            this.updateMessage();
          }}
        >
          Click!!
        </button>
      </>
    );
  }
}
```

You didn't use state in functional components because it was only supported in class components. But over the years hooks have been implemented in functional components which enables to use state too.

The `useState()` hook can used to implement state in functional components. It returns an array with two items: the first item is current state and the next one is a function (setState) that updates the value of the current state.

Let's see an example to demonstrate the state in functional components,

```
function App() {
  const [message, setMessage] = useState("This is a functional component");
  const updateMessage = () => {
    setMessage("Updating the functional component");
  };
  return (
    <div className="App">
      <h1>{message} </h1>
      <button onClick={updateMessage}>Click me!!</button>
    </div>
```

```
  );
}
```

3. Props:

Props are referred to as "properties". The props are passed into React component just like arguments passed to a function. In other words, they are similar to HTML attributes.

The props are accessible in child class component using `this.props` as shown in below example,

```
class Child extends React.Component {
  render() {
    return (
      <h1>
        {" "}
        This is a functional component and component name is {
          this.props.name
        }{" "}
      </h1>
    );
  }
}

class Parent extends React.Component {
  render() {
    return (
      <div className="Parent">
        <Child name="First child component" />
        <Child name="Second child component" />
      </div>
    );
  }
}
```

Props in functional components are similar to that of the class components but the difference is the absence of 'this' keyword.

```
function Child(props) {
  return (
    <h1>
      This is a child component and the component name is{props.name}
    </h1>
  );
}

function Parent() {
  return (
    <div className="Parent">
      <Child name="First child component" />
      <Child name="Second child component" />
    </div>
  );
}
```

245. 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 { StrictMode } from "react";

function App() {
  return (
    <div>
      <Header />
      <StrictMode>
        <div>
          <ComponentOne />
          <ComponentTwo />
        </div>
      </StrictMode>
      <Header />
    </div>
  );
}
```

In the example above, the *strict mode* checks apply to `<ComponentOne>` and `<ComponentTwo>` components only. i.e., Part of the application only.

Note: Frameworks such as NextJS has this flag enabled by default.

246. What is the benefit of strict mode?

The will be helpful in the below cases,

1. To find the bugs caused by impure rendering where the components will re-render twice.
2. To find the bugs caused by missing cleanup of effects where the components will re-run effects one more extra time.
3. Identifying components with **unsafe lifecycle methods**.
4. Warning about **legacy string ref** API usage.
5. Detecting unexpected **side effects**.
6. Detecting **legacy context** API.
7. Warning about deprecated **findDOMNode** usage

247. Why does strict mode render twice in React?

StrictMode renders components twice in development mode(not production) in order to detect any problems with your code and warn you about those problems. This is used to detect accidental side effects in the render phase. If you used `create-react-app` development tool then it automatically enables StrictMode by default.

```
const root = createRoot(document.getElementById("root"));
root.render(
  <StrictMode>
```

```
    <App />
  </StrictMode>
);
```

If you want to disable this behavior then you can simply remove `strict` mode.

```
const root = createRoot(document.getElementById("root"));
root.render(<App />);
```

To detect side effects the following functions are invoked twice:

1. Function component bodies, excluding the code inside event handlers.
2. Functions passed to `useState`, `useMemo`, or `useReducer` (any other Hook)
3. Class component's `constructor`, `render`, and `shouldComponentUpdate` methods
4. Class component static `getDerivedStateFromProps` method
5. State updater functions

248. What are the rules of JSX?

The below 3 rules needs to be followed while using JSX in a react application.

1. **Return a single root element**:
   If you are returning multiple elements from a component, wrap them in a single parent element. Otherwise you will receive the below error in your browser console.

   ```
   html Adjacent JSX elements must be wrapped in an enclosing tag.
   ```

2. **All the tags needs to be closed:**
   Unlike HTML, all tags needs to closed explicitly with in JSX. This rule applies for self-closing tags(like hr, br and img tags) as well.

3. **Use camelCase naming:**
   It is suggested to use camelCase naming for attributes in JSX. For example, the common attributes of HTML elements such as `class`, `tabindex` will be used as `className` and `tabIndex`.
   Note: There is an exception for `aria-*` and `data-*` attributes which should be lower cased all the time.

249. What is the reason behind multiple JSX tags to be wrapped?

Behind the scenes, JSX is transformed into plain javascript objects. It is not possible to return two or more objects from a function without wrapping into an array. This is the reason you can't simply return two or more JSX tags from a function without
wrapping them into a single parent tag or a Fragment.

250. How do you prevent mutating array variables?

The preexisting variables outside of the function scope including state, props and context leads to a mutation and they result in unpredictable bugs during the rendering stage. The below points should be taken care while working with arrays variables.

1. You need to take copy of the original array and perform array operations on it for the rendering purpose. This is called local mutation.

2. Avoid triggering mutation methods such as push, pop, sort and reverse methods on original array. It is safe to use filter, map and slice method because they create a new array.

251. What are capture phase events?

The `onClickCapture` React event is helpful to catch all the events of child elements irrespective of event propagation logic or even if the events propagation stopped. This is useful if you need to log every click events for analytics purpose.

For example, the below code triggers the click event of parent first followed by second level child eventhough leaf child button elements stops the propagation.

```
<div onClickCapture={() => alert("parent")}>
  <div onClickCapture={() => alert("child")}>
    <button onClick={(e) => e.stopPropagation()} />
    <button onClick={(e) => e.stopPropagation()} />
  </div>
</div>
```

The event propagation for the above code snippet happens in the following order:

1. It travels downwards in the DOM tree by calling all `onClickCapture` event handlers.
2. It executes `onClick` event handler on the target element.
3. It travels upwards in the DOM tree by call all `onClick` event handlers above to it.

252. How does React updates screen in an application?

React updates UI in three steps,

1. **Triggering or initiating a render:** The component is going to triggered for render in two ways.
    1. **Initial render:** When the app starts, you can trigger the initial render by calling `creatRoot` with the target DOM node followed by invoking component's `render` method. For example, the following code snippet renders `App` component on root DOM node.

```
import { createRoot } from "react-dom/client";

const root = createRoot(document.getElementById("root"));
root.render(<App />);
```

    2. **Re-render when the state updated:** When you update the component state using the state setter function, the componen't state automatically queues for a render.
2. **Rendering components:** After triggering a render, React will call your components to display them on the screen. React will call the root component for initial render and call the function component whose state update triggered the render. This is a recursive process for all nested components of the target component.
3. **Commit changes to DOM:** After calling components, React will modify the DOM for initial render using `appendChild()` DOM API and apply minimal necessary DOM updates for re-renders based on differences between rerenders.

253. How does React batch multiple state updates?

React prevents component from re-rendering for each and every state update by grouping multiple state updates within an event handler. This strategy improves the application performance and this process known as **batching**. The older version of React only supported batching for browser events whereas React18 supported for asynchronous actions, timeouts and intervals along with native events. This improved version of batching is called **automatic batching**.

Let's demonstrate this automatic batching feature with a below example.

```
import { useState } from "react";

export default function BatchingState() {
  const [count, setCount] = useState(0);
  const [message, setMessage] = useState("batching");

  console.log("Application Rendered");

  const handleUsers = () => {
    fetch("https://jsonplaceholder.typicode.com/users/1").then(() => {
      // Automatic Batching re-render only once
      setCount(count + 1);
      setMessage("users fetched");
    });
  };

  return (
    <>
      <h1>{count}</h1>
      <button onClick={handleAsyncFetch}>Click Me!</button>
    </>
  );
}
```

The preceding code updated two state variables with in an event handler. However, React will perform automatic batching feature and the component will be re-rendered only once for better performance.

254. Is it possible to prevent automatic batching?

Yes, it is possible to prevent automatic batching default behavior. There might be cases where you need to re-render your component after each state update or updating one state depends on another state variable. Considering this situation, React introduced `flushSync` method from `react-dom` API for the usecases where you need to flush state updates to DOM immediately.

The usage of `flushSync` method within an `onClick` event handler will be looking like as below,

```
import { flushSync } from "react-dom";

const handleClick = () => {
  flushSync(() => {
    setClicked(!clicked); //Component will create a re-render here
  });
```

```
  setCount(count + 1); // Component will create a re-render again here
};
```

In the above click handler, React will update DOM at first using flushSync and second time updates DOM because of the counter setter function by avoiding automatic batching.

255. What is React hydration?

React hydration is used to add client-side JavaScript interactivity to pre-rendered static HTML generated by the server. It is used only for server-side rendering(SSR) to enhance the initial rendering time and make it SEO friendly application. This hydration acts as a bridge to reduce the gap between server side and client-side rendering.

After the page loaded with generated static HTML, React will add application state and interactivity by attaching all event handlers for the respective elements. Let's demonstrate this with an example.

Consider that React DOM API(using `renderToString`) generated HTML for `<App>` component which contains `<button>` element to increment the counter.

```
import {useState} from 'react';
import { renderToString } from 'react-dom/server';

export default function App() {
  const [count, setCount] = React.useState(0);

  return (
    <h1>Counter</h1>
    <button onClick={() => setCount(prevCount => prevCount + 1)}>
      {count} times
    </button>
    );
}

const html = renderToString(<App />);
```

The above code generates the below HTML with a header text and button component without any interactivity.

```
<h1>Counter</h1>
<button>
  <!-- -->0<!-- -->
  times
</button>
```

At this stage `hydrateRoot` API can be used to perform hydration by attaching `onClick` event handler.

```
import { hydrateRoot } from "react-dom/client";
import App from "./App.js";

hydrateRoot(document.getElementById("root"), <App />);
```

After this step, you are able to run React application on server-side and hydrating the javascript bundle on client-side for smooth user experience and SEO purposes.

256. How do you update objects inside state?

You cannot update the objects which exists in the state directly. Instead, you should create a fresh new object (or copy from the existing object) and update the latest state using the newly created object. Eventhough JavaScript objects are mutable, you need to treat objects inside state as read-only while updating the state.

Let's see this comparison with an example. The issue with regular object mutation approach can be described by updating the user details fields of `Profile` component. The properties of `Profile` component such as firstName, lastName and age details mutated in an event handler as shown below.

```
import { useState } from "react";

export default function Profile() {
  const [user, setUser] = useState({
    firstName: "John",
    lastName: "Abraham",
    age: 30,
  });

  function handleFirstNameChange(e) {
    user.firstName = e.target.value;
  }

  function handleLastNameChange(e) {
    user.lastName = e.target.value;
  }

  function handleAgeChange(e) {
    user.age = e.target.value;
  }

  return (
    <>
      <label>
        First name:
        <input value={user.firstName} onChange={handleFirstNameChange} />
      </label>
      <label>
        Last name:
        <input value={user.lastName} onChange={handleLastNameChange} />
      </label>
      <label>
        Age:
        <input value={user.age} onChange={handleAgeChange} />
      </label>
      <p>
        Profile:
        {person.firstName} {person.lastName} ({person.age})
      </p>
    </>
  );
}
```

Once you run the application with above user profile component, you can observe that user profile details won't be update upon entering the input fields.
This issue can be fixed by creating a new copy of object which includes existing properties through spread syntax(...obj) and add changed values in a single event handler itself as shown below.

```
handleProfileChange(e) {
  setUser({
  ...user,
    [e.target.name]: e.target.value
  });
}
```

The above event handler is concise instead of maintaining separate event handler for each field. Now, UI displays the updated field values as expected without an issue.

257. How do you update nested objects inside state?

You cannot simply use spread syntax for all kinds of objects inside state. Because spread syntax is shallow and it copies properties for one level deep only. If the object has nested object structure, UI doesn't work as expected with regular JavaScript nested property mutation. Let's demonstrate this behavior with an example of User object which has address nested object inside of it.

```
const user = {
  name: "John",
  age: 32,
  address: {
    country: "Singapore",
    postalCode: 440004,
  },
};
```

If you try to update the country nested field in a regular javascript fashion(as shown below) then user profile screen won't be updated with latest value.

```
user.address.country = "Germany";
```

This issue can be fixed by flattening all the fields into a top-level object or create a new object for each nested object and point it to it's parent object. In this example, first you need to create copy of address object and update it with the latest value. Later, the address object should be linked to parent user object something like below.

```
setUser({
  ...user,
  address: {
    ...user.address,
    country: "Germany",
```

```
  },
});
```

This approach is bit verbose and not easy for deep hierarchical state updates. But this workaround can be used for few levels of nested objects without much hassle.

258. How do you update arrays inside state?

Eventhough arrays in JavaScript are mutable in nature, you need to treat them as immutable while storing them in a state. That means, similar to objects, the arrays cannot be updated directly inside state. Instead, you need to create a copy of the existing array and then set the state to use newly copied array.

To ensure that arrays are not mutated, the mutation operations like direct direct assignment(arr[1]='one'), push, pop, shift, unshift, splice etc methods should be avoided on original array. Instead, you can create a copy of existing array with help of array operations such as filter, map, slice, spread syntax etc.

For example, the below push operation doesn't add the new todo to the total todo's list in an event handler.

```
onClick = {
  todos.push({
    id: id+1,
    name: name
  })
}
```

This issue is fixed by replacing push operation with spread syntax where it will create a new array and the UI updated with new todo.

```
onClick = {
  [
    ...todos,
    { id: id+1, name: name }
  ]
}
```

259. How do you use immer library for state updates?

Immer library enforces the immutability of state based on **copy-on-write** mechanism. It uses JavaScript proxy to keep track of updates to immutable states. Immer has 3 main states as below,

1. **Current state:** It refers to actual state
2. **Draft state:** All new changes will be applied to this state. In this state, draft is just a proxy of the current state.
3. **Next state:** It is formed after all mutations applied to the draft state

Immer can be used by following below instructions,

1. Install the dependency using `npm install use-immer` command
2. Replace `useState` hook with `useImmer` hook by importing at the top
3. The setter function of `useImmer` hook can be used to update the state.

For example, the mutation syntax of immer library simplifies the nested address object of user state as follows,

```
import { useImmer } from "use-immer";
const [user, setUser] = useImmer({
  name: "John",
  age: 32,
  address: {
    country: "Singapore",
    postalCode: 440004,
  },
});

//Update user details upon any event
setUser((draft) => {
  draft.address.country = "Germany";
});
```

The preceding code enables you to update nested objects with a conceise mutation syntax.

260. What are the benefits of preventing the direct state mutations?

261. What are the preferred and non-preferred array operations for updating the state?

The below table represent preferred and non-preferred array operations for updating the component state.

Action	Preferred	Non-preferred
Adding	concat, [...arr]	push, unshift
Removing	filter, slice	pop, shift, splice
Replacing	map	splice, arr[i] = someValue
sorting	copying to new array	reverse, sort

If you use Immer library then you can able to use all array methods without any problem.

262. What will happen by defining nested function components?

Technically it is possible to write nested function components but it is not suggested to write nested function definitions. Because it leads to unexpected bugs and performance issues.

263. Can I use keys for non-list items?

Keys are primarily used for rendering list items but they are not just for list items. You can also use them React to distinguish components. By default, React uses order of the components in

264. What are the guidelines to be followed for writing reducers?

There are two guidelines to be taken care while writing reducers in your code.

1. Reducers must be pure without mutating the state. That means, same input always returns the same output. These reducers run during rendering time similar to state updater functions. So these functions should not send any requests, schedule time outs and any other side effects.

2. Each action should describe a single user interaction eventhough there are multiple changes applied to data. For example, if you "reset" registration form which has many user input fields managed by a reducer, it is suggested to send one "reset" action instead of creating separate action for each fields. The proper ordering of actions should reflect the user interactions in the browser and it helps a lot for debugging purpose.

265. What is useReducer hook? Can you describe its usage?

266. How do you compare useState and useReducer?

267. How does context works using useContext hook?

268. What are the use cases of useContext hook?

Some of the common use cases of useContext are listed below,

1. **Theme customizations:** The useContext hook can be used to manage and apply custom themes for an application. That means it allows users to personalize the appearance of the application.
2. **Support localization:** The context hook is helpful to implement localization by providing translated strings to components based on the user's language/locale preference.
3. **User authentication:** It can be used to manage user authentication or session status and display user specific information with in components.

269. When to use client and server components?

You can efficiently build nextjs application if you are aware about which part of the application needs to use client components and which other parts needs to use server components. The common cases of both client and server components are listed below:

Client components:

1. Whenever your need to add interactivity and event listeners such as onClick(), onChange(), etc to the pages
2. If you need to use State and Lifecycle Effects like useState(), useReducer(), useEffect() etc.
3. If there is a requirement to use browser-only APIs.
4. If you need to implement custom hooks that depend on state, effects, or browser-only APIs.
5. There are React Class components in the pages.

Server components:

1. If the component logic is about data fetching.
2. If you need to access backend resources directly.
3. When you need to keep sensitive information((access tokens, API keys, etc)) on the server.
4. If you want reduce client-side JavaScript and placing large dependencies on the server.

270. What are the differences between page router and app router in nextjs?

-- ----- ALSO GET DATA FROM THE TS LECTURE NOTES FROM (DAILYCODE + CMS + MY_NOTES)

- TypeScript is JavaScript with added syntax for types.
- TypeScript uses compile time type checking. Which means it checks if the specified types match before running the code, not while running the code.

- TypeScript is transpiled into JavaScript using a compiler.

```
# Type Assignment ->
# 1. Explicit: writing out the type:
let firstName: string = "Dylan";

# 2. Implicit: TypeScript will "guess" the type, based on the assigned value:
let firstName = "Dylan";

# Note: Having TypeScript "guess" the type of a value is called infer.
# TypeScript may not always properly infer what the type of a variable may be. In such cases, it will set the type to any which disables type checking.
```

- TypeScript Simple Types
    - There are three main primitives in JavaScript and TypeScript.
        - boolean - true or false values
        - number - whole numbers and floating point values
        - string - text values like "TypeScript Rocks"
    - There are also 2 less common primitives used in later versions of Javascript and TypeScript.
        - bigint - whole numbers and floating point values, but allows larger negative and positive numbers than the number type.
        - symbol are used to create a globally unique identifier.

```

```

- TypeScript Special Types - [These types don't have much use]
    - any - it disables type checking and effectively allows all types to be used.
        - any can be a useful way to get past errors since it disables type checking, but TypeScript will not be able to provide type safety, and tools which rely on type data, such as auto completion, will not work. Remember, it should be avoided at "any" cost...
    - unknown - it is a similar, but safer alternative to any.
        - unknown is best used when you don't know the type of data being typed. To add a type later, you'll need to cast it. Casting is when we use the "as" keyword to say property or variable is of the casted type.
    - never - it effectively throws an error whenever it is defined.
        - never is rarely used, especially by itself, its primary use is in advanced generics.
    - undefined & null - undefined and null are types that refer to the JavaScript primitives undefined and null respectively.

```
let y: undefined = undefined;
let z: null = null;
```

- TypeScript Arrays

```
const names: string[] = [];
names.push("Dylan"); # no error

# The readonly keyword can prevent arrays from being changed.
const names: readonly string[] = ["Dylan"];
names.push("Jack"); # Error: Property 'push' does not exist on type 'readonly string[]'.

# TypeScript can infer the type of an array if it has values.
const numbers = [1, 2, 3]; # inferred to type number[]
const numbers = [1, 2, 3, "a"]; # inferred to type (string | number)[]

numbers.push(4); # no error
```

- TypeScript Tuples
  - A tuple is a typed array with a pre-defined length and types for each index.

```
# define our tuple
let ourTuple: [number, boolean, string]; # CONST WILL GIVE ERROR FOR TUPLES [USE LET
ONLY]

ourTuple = [5, false, 'Coding']; # initialize correctly
ourTuple = [false, 'Coding God was mistaken', 5]; # wrong order will also gives error
console.log(ourTuple[0]); # To access 1st element

const ourTuple: [string, number] = ["s", 2,2,4]; # YOU CAN WRITE ANY TYPE AFTER THIS SO
CREATE READONLY TUPLES --NO IT IS GIVING ERROR

# READONLY TUPLES
# define our readonly tuple - YOU CANNOT CHANGE IT LATER
const ourReadonlyTuple: readonly [number, boolean, string] = [5, true, 'Coding'];

# useState in react returns a tuple of the value and a setter function. It's a example of
tuple : const [firstName, setFirstName] = useState('Dylan')

# NAMED TUPLES ::
# you can access elements by their names, making the code more readable.
const ourTuple: [letter: string, digit: number] = ["s", 2];

# Destructuring Tuples::
# Since tuples are arrays we can also destructure them.
const graph: [number, number] = [55.2, 41.3];
const [x, y] = graph;
```

- TypeScript Object Types

```
# OBJECT TYPE
```

```
const car: { type: string, model: string, year: number } = {
  type: "Toyota", model: "Corolla", year: 2009 };

# TypeScript can infer the types of properties based on their values.
const ourTuple = { name: "ok" };
ourTuple.name = "go"; # Can change without error
ourTuple.name = 3; # will show error

# Optional Properties::

const car: { type: string, mileage: number } = { # error
  type: "Toyota"
};
const car: { type: string, mileage?: number } = {
# added "?" so mileage property is now optional so no error
  type: "Toyota"
};
car.mileage = 2000;

# Index Signatures::

# this means the key is string and value is number for all the value
const obj: { [index: string]: number } = { one: 1 };
obj.two = 2; # Works fine
obj.three = "three"; # Gives Error

# Index signatures like this one can also be expressed with utility types like
Record<string, number>.
```

- TypeScript Enums
    - An enum is a special "class" that represents a group of constants (unchangeable variables).
    - Enums come in two flavors string and numeric. Lets start with numeric.

```
# Numeric Enums - Default::

# By default, enums will initialize the first value to 0 and add 1 to each additional
value:
enum CardinalDirections {
  North,
  East,
  South,
  West,
}
console.log(CardinalDirections.North);  # 0
console.log(CardinalDirections.East);   # 1
console.log(CardinalDirections.["South"]);  # 2 # You can also call it this way
console.log(CardinalDirections.["West"]);   # 3
console.log(CardinalDirections[1]);        # East
console.log(CardinalDirections[3]);        # West
console.log(CardinalDirections[0]);        # North

# Numeric Enums - Initialized::

# If no value is provided then it will add one to next enum value
```

```
enum CardinalDirections {
  North = 5,
  East,
  South,
  West,
}
console.log(CardinalDirections.North);  # 5
console.log(CardinalDirections.East);   # 6
console.log(CardinalDirections.South);  # 7
console.log(CardinalDirections.West);   # 8

# Enum can have duplicate values but avoid doing this
enum CardinalDirections {
  North = 5,     # 5
  East,          # 6
  South = 5,     # 5
  West,          # 6
}
# Numeric Enums - Fully Initialized::

enum StatusCodes {
  NotFound = 404,
  Success = 200,
  Accepted = 202,
  BadRequest = 400,
}
console.log(StatusCodes["Accepted"]);    # 404 # You can also write it this way
console.log(StatusCodes["Success"]);     # 200
console.log(StatusCodes.Accepted);       # 202
console.log(StatusCodes.BadRequest);     # 400
console.log(StatusCodes[200]);           # Success

# String Enums ::

# Technically, you can mix and match string and numeric enum values, but it is
recommended not to do so.
enum CardinalDirections {
  North = "myNorth",
  East = "myEast",
  South = "mySouth",
  West = "myWest",
}
console.log(CardinalDirections["North"]);     # myNorth
console.log(CardinalDirections["East"]);      # myEast
console.log(CardinalDirections.South);        # mySouth
console.log(CardinalDirections.West);         # myWest
console.log(CardinalDirections["MyWest"]);    # West  #YOU CAN'T CALL BY VALUES (CAN DO
IN NUM)
```

- TypeScript "Type" Aliases and "Interfaces"
    - Aliases and Interfaces allows types to be easily shared between different variables/objects.

```
1. Type - # allow defining types with a custom name
# Type Aliases can be used for primitives like string or more complex types such as
```

```
objects and arrays:

# First define the types here
type CarYear = number;
type CarType = string;
type CarModel = string;
type Car = {
  year: CarYear; # Year is number as defined by CarYear
  type: CarType;
  model: CarModel;
};

# Then just use name here
const myCarYear: CarYear = 2001;
const myCarType: CarType = "Toyota";
const myCarModel: CarModel = "Corolla";
const myCar: Car = {
  year: myCarYear,
  type: myCarType,
  model: myCarModel,
};

# Extending types :: - [MEANS ADDING TYPES]
# It means creating a new interface with the same properties as the original, plus
something new.
type Name = {
  name: string;
};
type Age = {
  age: number;
};
type Person = Name & Age; # You can add them with "&"
const person: Person = {
  name: "Alice",
  age: 30
};
console.log(person); # { name: "Alice", age: 30 }

2. Interfaces
# Interfaces are similar to type aliases, except they only apply to object types.

interface Rectangle {
  height: number,
  width: number
}

const rectangle: Rectangle = {
  height: 20,
  width: 10
};

# Extending Interfaces ::
# It means creating a new interface with the same properties as the original, plus
something new.
interface Rectangle {
  height: number;
  width: number;
}
```

```
interface ColoredRectangle extends Rectangle {
  color: string;
  // height: string; # gives error as it contradicts with the extend type {can write same
type}
}
const coloredRectangle: ColoredRectangle = {
  height: 20,
  width: 10,
  color: "red"
};

# Declaration Merging :: -> ONLY INTERFACE CAN DO IT AND NOT TYPES
# you can define multiple declarations with the same name, and TypeScript will
automatically merge them into a single interface.
interface Person {
  name: string;
}
interface Person {
  age: number;
}
const person: Person = {
  name: "Alice",
  age: 30,
};

*** Differences Between interface and type in TypeScript:
    i- Declaration Merging:
        - Interface: Supports declaration merging.
        - Type: Does not support declaration merging.

    ii- Usage with Classes:
        - Interface: Can be implemented by classes.
        - Type: Can describe class instances but is less idiomatic for class
implementation.

    iii- Extending/Intersection:
        - Interface: Uses extends for inheritance.
        - Type: Uses intersection types (&) for combining types.

      4. Function Overloads:
        - Interface: Can define multiple function signatures (overloads).
        - Type: Can define function types but less naturally supports overloads.

      5. Recursive Types:
        - Interface: Naturally suited for recursive structures.
        - Type: Can handle recursive structures but with different syntax.
```

- TypeScript Union Types (Union '|' (OR))
  - Union types are used when a value can be more than a single type.
  - Such as when a property would be string or number.

```
# Using the | we are saying our parameter is a string or number
function printStatusCode(code: string | number) {
  console.log(`My status code is ${code}.`);
```

```
}
printStatusCode(404);
printStatusCode("404");

# Union Type Errors
# Note: you need to know what your type is when union types are being used to avoid type
errors:
function printStatusCode(code: string | number) {
  console.log(`My status code is ${code.toUpperCase()}.`);
}
# The above line will give error as toUpperCase() property is not for numbers:: YOU CAN
SOLVE THIS BY USING "as" (CASTING) FOR PERTICULAR VARIABLE TO SPECIFY THE TYPE
function printStatusCode(code: string | number) {
  console.log(`My status code is ${(code as string).toUpperCase()}.`);
}
```

- TypeScript Functions

```
1. Return Type

# What type of value this function will return - MOST OF THE TIME IT WILL INFER
function getTime(): number { # REMOVING RETURN TYPE THE FUNCTION WILL INFER THE TYPE HERE
  return new Date().getTime();
}
# If no return type is defined, TypeScript will attempt to infer it through the types of
the variables or expressions returned.

# Void Return Type ::
function printHello(): void {
  console.log('Hello!');
}

2. Parameters

# Function parameters are typed with a similar syntax as variable declarations.
function multiply(a: number, b: number) {
  return a * b;
}

# Optional Parameters :: "?"

# By default TypeScript will assume all parameters are required, but they can be
explicitly marked as optional.
// the `?` operator here marks parameter `c` as optional
function add(a: number, b: number, c?: number) {
  return a + b + (c || 0);
}

# Default Parameters :: "="

# The default value goes after the type annotation:
function pow(value: number, exponent: number = 10) {
  return value ** exponent;
```

```
}

# Named Parameters :: "{}:{}"

function divide({ dividend, divisor }: { dividend: number, divisor: number }) {
  return dividend / divisor;
}

# YOU CAN CREATE TYPE FOR NAMED PARAMETERS THIS WAY
type Finding2 = (p: { x: string; y: number }) => number;

# Rest Parameters :: "..."

# Rest parameters can be typed like normal parameters, but the type must be an array as
rest parameters are always arrays.
function add(a: number, b: number, ...rest: number[]) {
  return a + b + rest.reduce((p, c) => p + c, 0);
}

3. Type Alias for functions

# Function types can be specified separately from functions with type aliases.
# These types are written similarly to arrow functions
# ONLY USED FOR FUNCTION EXPRESSION(VARIABLE FN) AND NOT FUNCITON DECLARATION(NORMAL FN)
# FOR FUNCTION DECLARATION USE INLINE TYPES OR MAY USE UTILITY TYPES (BUT AVOID THIS)

type FuncitonType = (age: number) => String;

const addAge: FuncitonType = (age) => {
  // return 3; # This will give error as return type is specified to string and not
number
  return `Your age is ${age}`;
};
```

- TypeScript Casting
  - Casting is the process of overriding a type.

```
1. Casting with as

let x: unknown = "hello";
console.log((x as string).length);

# Casting doesn't actually change the type of the data within the variable
let x: unknown = 1;
console.log((x as string).length); # returns undefined (as won't change 1 into "1")
// console.log((2 as string).length); # This will give error as it's a number directly

# Force casting
# To override type errors that TypeScript may throw when casting, first cast to unknown,
then to the target type.
console.log((2 as unknown as string).length); # convert number to unknown first (if
intentional)-> STILL GIVE ERROR (JAVASCRIPT) CANNOT USE LENGTH AS IN NUMBER
```

```
2. Casting with <>

const x: number | string = "3";
# below both are same
console.log((x as string).toUpperCase());
console.log((<string>x).toUpperCase());

#  Here x is number so type conversion will give error (to solve this first convert them
into unknown)
const x: number | string = 3;
// console.log((x as string).toUpperCase()); # Error TYPESCRIPT
// console.log((<string>x).toUpperCase()); # Error TYPESCRIPT
console.log((x as unknown as string).toUpperCase()); # Error JAVASCRIPT ( NO TYPESCRIPT)
console.log((<string>(<unknown>x)).toUpperCase()); # Error JAVASCRIPT ( NO TYPESCRIPT)
```

- TypeScript Classes
    - TypeScript adds types and visibility modifiers to JavaScript classes.

```
1. Members: Types

# The members of a class (properties & methods) are typed using type annotations, similar
to variables.
class Person {
  name: string;
}
const person = new Person();
person.name = "Jane";

2. Members: Visibility

# There are three main visibility modifiers in TypeScript.
    #1. public - (default) allows access to the class member from anywhere
    #2. private - only allows access to the class member from within the class
    #3. protected - allows access to the class member from itself and any classes that
inherit it

class Testing {
  public name: string; # You need to write this to write it in construction
  private age: number; # this can't be access outsite this class
  public constructor(name: string, age: number) { # NO NEED TO WRITE PUBLIC AS IT'S
DEFAULT
    this.name = name;
    this.age = age;
  }
  public get getInfo1() { # NO NEED TO WRITE PUBLIC AS IT'S DEFAULT
    return `My name is ${this.name}, and my age is ${this.age}`;
  }
  public getInfo2() {
    return `My name is ${this.name}, and my age is ${this.age}`;
  }
  private getInfo3() {
    return `My name is ${this.name}, and my age is ${this.age}`;
  }
}
```

```
const myName = new Testing("Mahesh", 28);
// console.log(myName.age);  # age is private so can't be call here
console.log(myName.name);
console.log(myName.getInfo1);
console.log(myName.getInfo2());
// console.log(myName.getInfo3()); # getInfo3 is private so can't be called here


# Parameter Properties
# TypeScript provides a convenient way to define class members in the constructor, by
adding a visibility modifiers to the parameter.

# This line declares and initialize the properties -> This line is same as below 5 lines
public constructor(private name: string, public age: number) {}

# NO NEED TO WRITE THIS 5 LINES NOW
    # public name: string;
    # private age: number;
    # public constructor(name: string, age: number) {
    # this.name = name;
    # this.age = age;

3. Readonly
# readonly keyword can prevent class members from being changed.

class Person {
  private readonly name: string; # This is readonly

  public constructor(name: string) {
    # name cannot be changed after this initial definition, which has to be either at
it's declaration or in the constructor.
    this.name = name;
  }
  public getName(): string {
    return this.name;
  }
  set changeName(newName: string) {
    // this.name = newName; # This will give error as name property is readonly
  }
}
const person = new Person("Jane");
console.log(person.getName());

4. Inheritance: Implements
# Interfaces can be used to define the type a class must follow through the implements
keyword.

interface GetNameType {
  getName: () => string; # the getName method will return string. It will be in
GetNameType
  greet: () => string;
}
type NewGreet = { # You can use type or interface here
  morningGreet: () => string;
}

# Use implements to add interface and use ,(comma) to add multiple interface
class Naming implements GetNameType, NewGreet {
```

```
  constructor(protected readonly name: string) {} # Use protected as it needed in
different class
  getName(): string { # Can skip string as it gets infer
    return this.name;
  }
  greet() {
    return `Hello!, ${this.name}`;
  }
  morningGreet() {
    return `Morning, ${this.name}`;
  }
}
const myName = new Naming("Mahesh");

console.log(myName.getName());      # Mahesh
console.log(myName.greet());        # Hello!, Mahesh
console.log(myName.morningGreet()); # Morning, Mahesh

5. Inheritance: Extends
# Classes can extend each other through the extends keyword.
# A class can only extends ONE OTHER CLASS.
6. Override
# When a class extends another class, it can replace the members of the parent class with
the same name.
# The override function is default (means you can change the parent function without
writing the override keyword). To force it to be used when overriding, Use the setting
noImplicitOverride in tsconfig(maybe)

# You can avoid such types as TS can infer it
interface GetSquare {
  fullInfo: () => string;
}

class Square extends Naming implements GetSquare {
  constructor(private readonly length: number, name: string) {
    super(name); # To add property from extended class use super
  }
  # The getName() function has override the function(with same name) from extended Naming
class
  override getName(): string {
    return ` Overriding the getName() function from Naming class`;
  }

  morningGreet() {
    return `Morning, ${this.name}`;
  }
  get squareValue() {
    return this.length * this.length;
  }
  fullInfo() {
    return `total length is ${this.squareValue}`;
  }
}

const myName = new Square(12, "Mahesh");
console.log(myName.getName());
console.log(myName.morningGreet());
console.log(myName.squareValue);
```

```
console.log(myName.fullInfo());

7. Abstract Classes
# Classes can be written in a way that allows them to be used as a base class for other
classes without having to implement all the members. This is done by using the abstract
keyword. Members that are left unimplemented also use the abstract keyword.

# NOT DOING IT NOW..
```

###TypeScript Basic Generics

- Generics in TypeScript allow you to create reusable components that work with a variety of types rather than a single type.
- They provide a way to create functions, classes, and interfaces that can operate with different data types while maintaining type safety.
- They enable you to write flexible and reusable code.
- Generics: Allow for type-safe and reusable code components.
- Generic Functions: Define functions that work with any type.
- Generic Classes: Create classes that operate with various types.
- Generic Interfaces and types: Describe objects with properties of different types.
- Generic Constraints: Restrict generics to types with specific properties.

```
1. Generic Functions # Define functions that work with any type.

function createPair<S, T>(v1: S, v2: T): [S, T] {
  return [v1, v2];
}
console.log(createPair<string, number>("hello", 42));
console.log(createPair<number, number>(2, 42));
console.log(createPair<string, boolean>("Are you good", false));
console.log(createPair("ok", 2)); # WORKS FINE, Type can be infer by typescript
console.log(createPair("ok", 2, 3)); # Will give error as only 2 parameters are allowed
console.log(createPair("ok")); # Will give error as only 2 parameters are allowed

2. Generic Classes # Create classes that operate with various types.

class NamedValue<T> {
  private _value: T | undefined;

  constructor(private name: string) {}

  public setValue(value: T) {
    this._value = value;
  }
  public getValue(): T | undefined {
    return this._value;
  }
```

```
  public toString(): string {
    return `${this.name}: ${this._value}`;
  }
}

let value = new NamedValue<number>("myNumber");
let value2 = new NamedValue<string>("myNumber");
value.setValue(10);
value2.setValue("OK");
console.log(value.toString()); // myNumber: 10
console.log(value2.toString()); // myNumber: OK

3. Generic Interfaces and types # Describe objects with properties of different types.

type ObjType<T, U> = { name: T; age: U };
interface NewObjType<S, T> {
  firstName: S;
  lastName: S;
  age: T;
}

const obj: ObjType<string, number> = { name: "Mahesh", age: 3 };
const newObj: NewObjType<string, number> = {
  firstName: "Mahesh",
  lastName: "Kumar",
  age: 28,
};

# Default Value ::

# Generics can be assigned default types which apply if no other value is "specified" or
"inferred".

type ObjType<T = string, U = number> = { name: T; age: U };
const obj: ObjType = { name: "Mahesh", age: 3 };
const obj: ObjType<string, string> = { name: "Mahesh", age: "three" }; # you can change
the default

4. Generic Constraints(Extends)
# Constraints can be added to generics to limit what's allowed.
# The constraints make it possible to rely on a more specific type when using the generic
type.

# example 1
function createLoggedPair<S extends string | number, T extends string | number>(
  v1: S,
  v2: T
): [S, T] {
  console.log(`creating pair: v1='${v1}', v2='${v2}'`);
  return [v1, v2];
}
createLoggedPair("Hello", "Fine");
createLoggedPair("Hello", 101);
// createLoggedPair("Hello", true); # Error as boolean doesn't exists on type

#Example 2
# Constrain T to types that have a length property (IF HAVE x.length property)
function logLength<T extends { length: number }>(arg: T): void {
```

```
  console.log(arg.length);
}
logLength("Hello");        # Works, string has a length property
logLength([1, 2, 3]);      # Works, array has a length property
// logLength(42);          # Error: number doesn't have a length property

# CAN ADD INTERFACE/TYPES FOR IT
interface Lengthwise {
  length: number;
}
function logLength<T extends Lengthwise>(arg: T): void {
  console.log(arg.length);
}
logLength("Hello");
```

- TypeScript Utility Types
  - TypeScript comes with a large number of types that can help with some common type manipulation, usually referred to as utility types.
  - [Partial, Required, Record, Omit, Pick, Exclude, ReturnType, Parameters, ReadOnly]

```
1. Partial # It changes all the properties in an object to be optional.

interface NewId {
  one: number;
  two: number;
}
const obj1: NewId = { one: 2, two: 4 };
const obj2: Partial<NewId> = { one: 2 }; # Now all obj are optional

2. Required # It changes all the properties in an object to be required.

interface NewId {
  one?: number;
  two?: number;
}
const obj1: NewId = { one: 2 };
const obj2: Required<NewId> = { one: 2, two: 4 }; # All are required/compulsory

3. Record # It is a shortcut to defining an object type with a specific key type and
value type.
# Record<string, number> is equivalent to { [key: string]: number } (tuples)
const nameAgeMap: Record<string, number> = {
  'Alice': 21,
  'Bob': 25
};

4. Omit # It removes keys from an object type. -> [only for object type]

interface Person {
  name: string;
  age: number;
  location?: string;
}
# age and location are removed from the type
const bob: Omit<Person, "age" | "location"> = { name: "Bob" };
```

```
5. Pick # It removes all but the specified keys from an object type.

interface Person {
  name: string;
  age: number;
  location?: string;
}
# Except age all other properties are removed
const bob: Pick<Person, "age"> = { age: 23 };

6. Exclude # It removes types from a union.

type Primitive = string | number | boolean;
const value1: Exclude<Primitive, string> = true;
const value2: Exclude<Primitive, string> = 234;
// const value3: Exclude<Primitive, string> = "abc"; # It will give error as string has
been excluded from the union type

7. ReturnType # It extracts the return type of a function type.

type PointGenerator = () => { x: number; y: number };
# It will give type of what a function will return
const point: ReturnType<PointGenerator> = {
  x: 10,
  y: 20,
};
console.log(point);

8. Parameters # It extracts the parameter types of a function type as an array.

type Finding = (x: string, y: number) => number;
type Finding2 = (p: { x: string; y: number }) => number;

const value1: Parameters<Finding> = ["name", 8];
const value2: Parameters<Finding2> = [{ x: "name", y: 8 }];

# How both are different in writing a function
const find: Finding = (x, y) => {
  return y + x.length;
};
const find2: Finding2 = (p) => {
  return p.y + p.x.length;
};
const find3: Finding2 = ({ x, y }) => {
  return y + x.length;
};

console.log(find("hello", 5)); // Output: 10
console.log(find2({ x: "hello", y: 5 })); // Output: 10
console.log(find3({ x: "hello", y: 5 })); // Output: 10

9. Readonly
# It is used to create a new type where all properties are readonly, meaning they cannot
be modified once assigned a value.
# Keep in mind TypeScript will prevent this at compile time, but in theory since it is
compiled down to JavaScript you can still override a readonly property.
interface Person {
```

```
  name: string;
  age: number;
}
const person: Readonly<Person> = {
  name: "Dylan",
  age: 35,
};
// person.name = "Israel"; # This will give error as it's readonly
```

- TypeScript Keyof
    - It is a keyword in TypeScript which is used to extract the key type from an object type.

```
1. keyof with explicit keys
# It is used to create a union type of the keys of a given object type.

# Example 1
interface Person {
  name: string;
  age: number;
  location: string;
}
type PersonKeys = keyof Person; # 'name' | 'age' | 'location'

const myPerson: PersonKeys = "age"; # You can only write any one value from the three

# Example 2
interface Person {
  name: string;
  age: number;
}
# `keyof Person` means "name" | "age"
function printPersonProperty(person: Person, property: keyof Person) {
  console.log(`Printing person property ${property}: "${person[property]}"`);
}
let person = {
  name: "Max",
  age: 27,
};
printPersonProperty(person, "name"); # Can only write "name" or "age" here

2. keyof with index signatures ->[NOT IMP MAY BE]
# keyof can also be used with index signatures to extract the index type.

type StringMap = { [key: string]: unknown };
# `keyof StringMap` resolves to `string` here
function createStringPair(property: keyof StringMap, value: string): StringMap {
  return { [property]: value };
}
console.log(createStringPair(22, "OOOOKKK")); // { '22': 'OOOOKKK' }
console.log(createStringPair("abc", "OOOOKKK")); // { '22': 'OOOOKKK' }
// console.log(createStringPair(true, "OOOOKKK")); // Error

# key must be 'string', 'number', 'symbol' and that too gets converted into string(?)
# In JavaScript, when you use a number as an object key, it is internally converted to a
```

```
string. This behavior causes TypeScript to include both string and number in the keyof
type.
```

- TypeScript Null & Undefined
    - By default null and undefined handling is disabled, and can be enabled by setting strictNullChecks to true.

```
# null and undefined are primitive types and can be used like other types, such as
string.
let value: string | undefined | null = null;
value = 'hello';
value = undefined;
```