1. What is React?

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

2. What are the major features of React?

The major features of React are:

- It uses **VirtualDOM** instead of RealDOM considering that RealDOM manipulations are expensive.
- Supports **server-side rendering**.
- Follows **Unidirectional** data flow or data binding.
- Uses **reusable/composable** UI components to develop the view.

3. What is JSX?

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

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

```
class App extends React.Component {
  render() {
    return(
      <div>
        <h1>{'Welcome to React world!'}</h1>
      </div>
    )
  }
}
```

4. What is the difference between Element and Component?

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

The object representation of React Element would be as follows:

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

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

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

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

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

Whereas a **component** can be declared in several different ways. It can be a class with a render() method or it can be defined as a function. In either case, it takes props as an input, and returns a JSX tree as the output:

```
const Button = ({ onLogin }) =>
  <div id={'login-btn'} onClick={onLogin}>Login</div>
```

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

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

5. How to create components in React?

There are two possible ways to create a component.

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

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


        }
```

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

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

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

If the component needs *state or lifecycle methods* then use class component otherwise use function component. *However, from React 16.8 with the addition of Hooks, you could use state , lifecycle methods and other features that were only available in class component right in your function component.* *So, it is always recommended to use Function components, unless you need a React functionality whose Function component equivalent is not present yet, like Error Boundaries *

7. What are Pure Components?

React.PureComponent is exactly the same as *React.Component* except that it handles the shouldComponentUpdate() method for you. When props or state changes, *PureComponent* will do a shallow comparison on both props and state. *Component* on the other hand won't compare current props and state to next out of the box. Thus, the component will re-render by default whenever shouldComponentUpdate is called.

8. What is state in React?

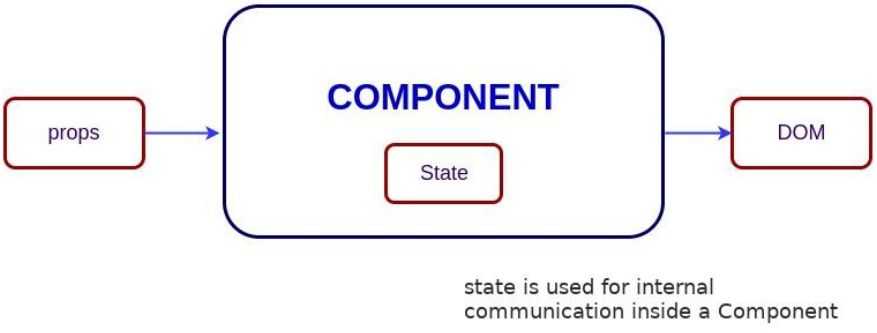
State of a component is an object that holds some information that may change over the lifetime of the component. We should always try to make our state as simple as possible and minimize the number of stateful components.

Let's create a user component with message state,

```
class User extends React.Component {
  constructor(props) {
    super(props)

    this.state = {
      message: 'Welcome to React world'
    }
  }

  render() {
    return (
      <div>
        <h1>{this.state.message}</h1>
      </div>
    )
  }
}
```



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.

9. What are props in React?

Props are inputs to components. They are single values or objects containing a set of values that are passed to components on creation using a naming convention similar to HTML-tag attributes. They are data passed down from a parent component to a child component.

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

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

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

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

This reactProp (or whatever you came up with) name then becomes a property attached to React's native props object which originally already exists on all components created using React library.

```
props.reactProp
```

10. What is the difference between state and props?

Both *props* and *state* are plain JavaScript objects. While both of them hold information that influences the output of render, they are different in their functionality with respect to component. Props get passed to the component similar to function parameters whereas state is managed within the component similar to variables declared within a function.

11. Why should we not update the state directly?

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

```
//Wrong
this.state.message = 'Hello world'
```

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

```
//Correct
this.setState({ message: 'Hello World' })
```

Note: You can directly assign to the state object either in *constructor* or using latest javascript's class field declaration syntax.

12. What is the purpose of callback function as an argument of setState()?

The callback function is invoked when setState finished and the component gets rendered. Since setState() is **asynchronous** the callback function is used for any post action.

Note: It is recommended to use lifecycle method rather than this callback function.

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

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

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

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

```
<button onclick='activateLasers()'>
```

Whereas in React it follows *camelCase* convention:

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

ii. 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.')
}
```

iii. 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)

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

There are 3 possible ways to achieve this:

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

```
class Foo extends Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    console.log('Click happened');
  }
  render() {
    return <button onClick={this.handleClick}>Click Me</button>;
  }
        }
```

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

```
handleClick = () => {
  console.log('this is:', this)
        }

        <button onClick={this.handleClick}>
        {'Click me'}
        </button>
```

iii. **Arrow functions in callbacks:** You can use *arrow functions* directly in the callbacks.

```
handleClick() {
    console.log('Click happened');
}
render() {
    return <button onClick={() => this.handleClick()}>Click Me</button>;
        }
```

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

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

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

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

This is an equivalent to calling .bind:

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

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

```
<button onClick={this.handleClick(id)} />
handleClick = (id) => () => {
  console.log("Hello, your ticket number is", id)
};
```

16. 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.

17. 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>
  }
```

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

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

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

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

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

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

Note:

i. 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.
ii. If you extract list item as separate component then apply *keys* on list component instead of li tag.
iii. There will be a warning message in the console if the key prop is not present on list items.

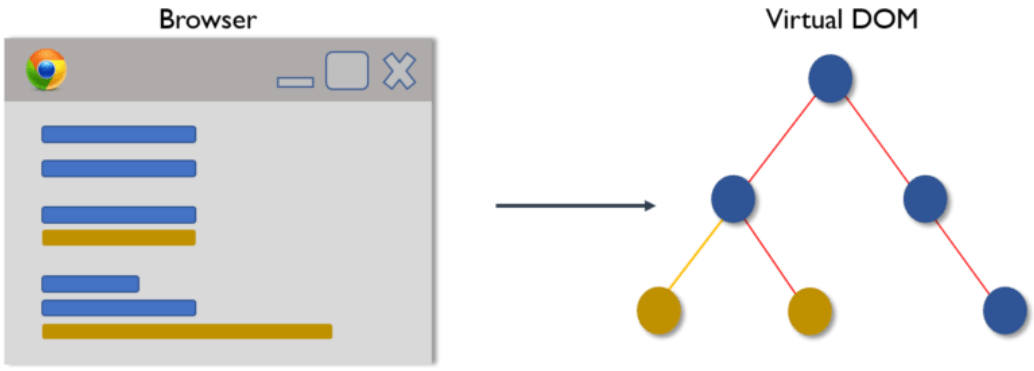
19. What is Virtual DOM?

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*.

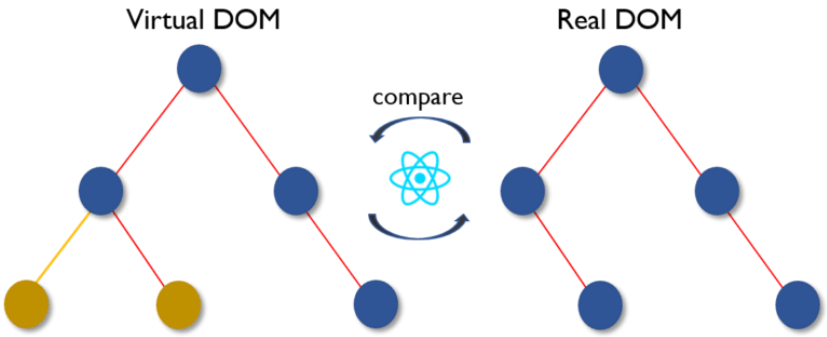
20. How Virtual DOM works?

The *Virtual DOM* works in three simple steps.

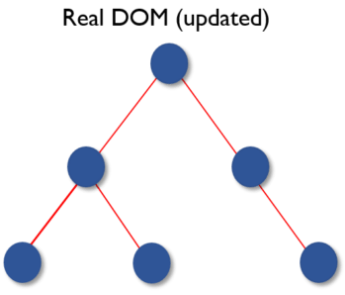
i. Whenever any underlying data changes, the entire UI is re-rendered in Virtual DOM representation.



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



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



21. 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.

22. What are controlled components?

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

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

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

23. What are uncontrolled components?

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

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

```
class UserProfile extends React.Component {
 constructor(props) {
  super(props)
  this.handleSubmit = this.handleSubmit.bind(this)
  this.input = React.createRef()
 }

 handleSubmit(event) {
  alert('A name was submitted: ' + this.input.current.value)
  event.preventDefault()
 }

 render() {
  return (
   <form onSubmit={this.handleSubmit}>
    <label>
```

```
          {'Name:'}
          <input type="text" ref={this.input} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

In most cases, it's recommend to use controlled components to implement forms. 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.

24. What are the different phases of component lifecycle?

The component lifecycle has three distinct lifecycle phases:

i. **Mounting:** The component is ready to mount in the browser DOM. This phase covers initialization from constructor(), getDerivedStateFromProps(), render(), and componentDidMount() lifecycle methods.

ii. **Updating:** In this phase, the component gets updated in two ways, sending the new props and updating the state either from setState() or forceUpdate(). This phase covers getDerivedStateFromProps(), shouldComponentUpdate(), render(), getSnapshotBeforeUpdate() and componentDidUpdate() lifecycle methods.

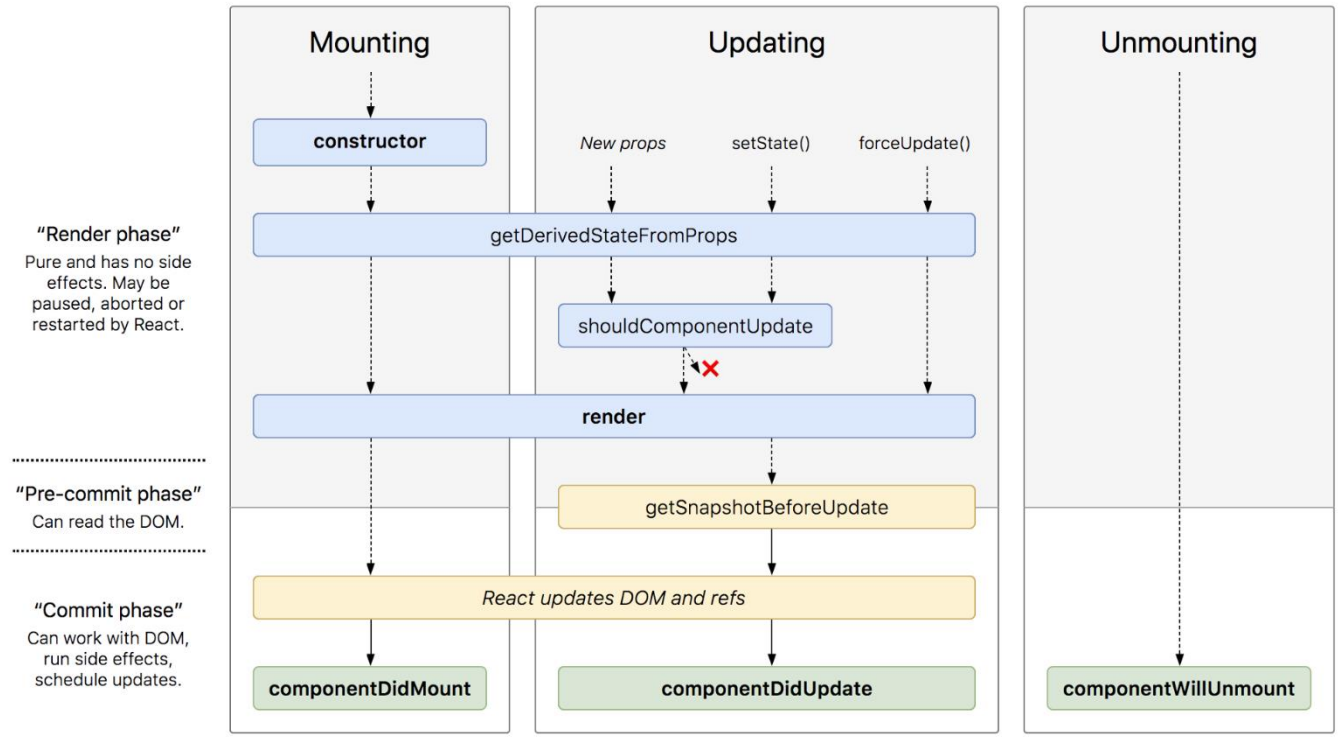
iii. **Unmounting:** In this last phase, the component is not needed and gets unmounted from the browser DOM. This phase includes componentWillUnmount() lifecycle method.

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

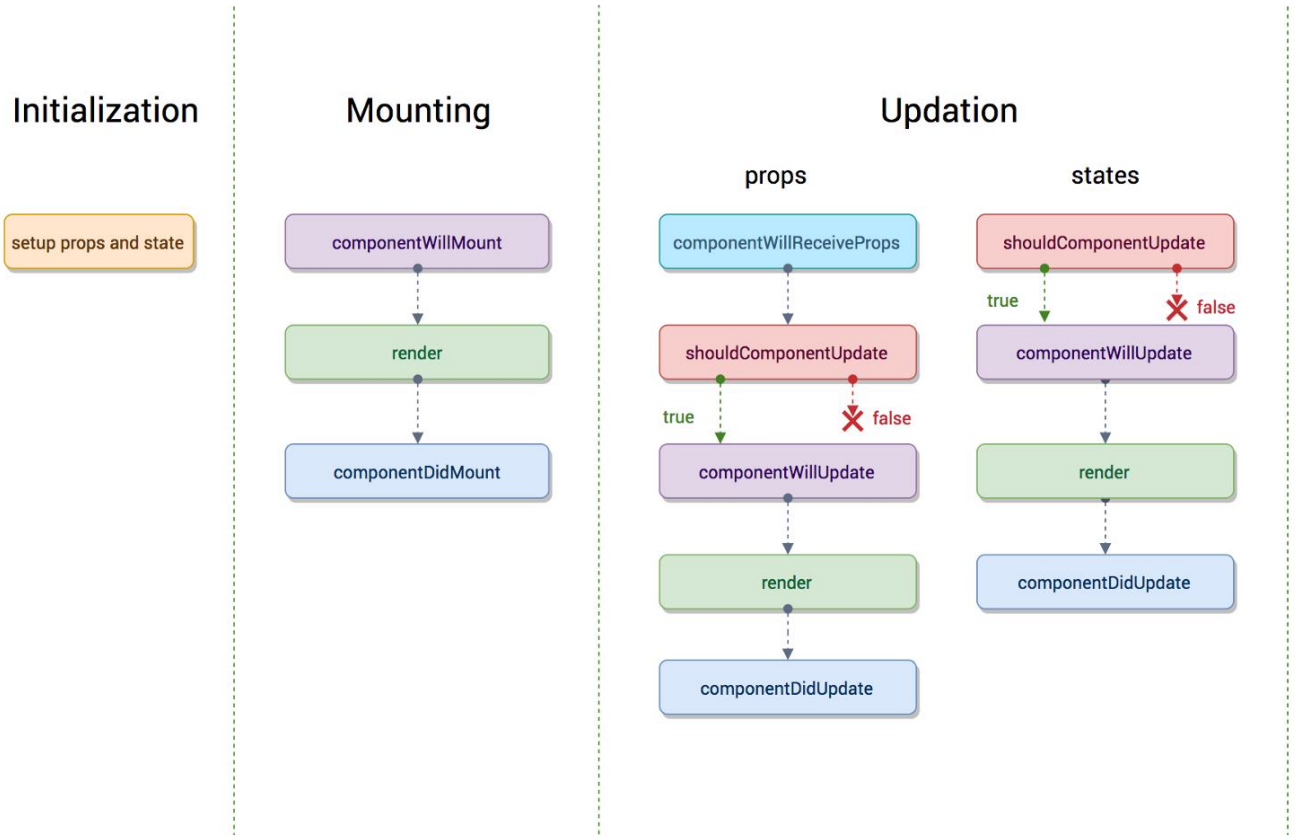
iv. **Render** The component will render without any side effects. This applies to Pure components and in this phase, React can pause, abort, or restart the render.

v. **Pre-commit** Before the component actually applies the changes to the DOM, there is a moment that allows React to read from the DOM through the getSnapshotBeforeUpdate().

vi. **Commit** React works with the DOM and executes the final lifecycles respectively componentDidMount() for mounting, componentDidUpdate() for updating, and componentWillUnmount() for unmounting.



Before React 16.3



25. What are the lifecycle methods of React?

Before React 16.3

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

React 16.3+

- **getDerivedStateFromProps:** Invoked right before calling render() and is invoked on *every* render. This exists for rare use cases where you need a derived state. Worth reading if you need derived state.
- **componentDidMount:** Executed after first rendering and where all AJAX requests, DOM or state updates, and set up event listeners should occur.
- **shouldComponentUpdate:** Determines if the component will be updated or not. By default, it returns true. If you are sure that the component doesn't need to render after the state or props are updated, you can return a false value. It is a great place to improve performance as it allows you to prevent a re-render if component receives a new prop.
- **getSnapshotBeforeUpdate:** Executed right before rendered output is committed to the DOM. Any value returned by this will be passed into componentDidUpdate(). This is useful to capture information from the DOM i.e. scroll position.

- **componentDidUpdate:** Mostly it is used to update the DOM in response to prop or state changes. This will not fire if shouldComponentUpdate() returns false.
- **componentWillUnmount** It will be used to cancel any outgoing network requests, or remove all event listeners associated with the component.

26. 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:

i. Code reuse, logic and bootstrap abstraction.
ii. Render hijacking.
iii. State abstraction and manipulation.
iv. Props manipulation.

27. What is context?

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

For example, authenticated users, locale preferences, UI themes need to be accessed in the application by many components.

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

28. What is children prop?

Children is a prop (this.props.children) 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.

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

A simple usage of children prop looks as below,

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

ReactDOM.render(
  <MyDiv>
    <span>{'Hello'}</span>
    <span>{'World'}</span>
  </MyDiv>,
  node
)
```

29. 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>
```

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

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

Passing props:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props)

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

Not passing props:

```
class MyComponent extends React.Component {
  constructor(props) {
    super()

    console.log(this.props) // prints undefined

    // but props parameter is still available
    console.log(props) // prints { name: 'John', age: 42 }
  }

  render() {
    // no difference outside constructor
    console.log(this.props) // prints { name: 'John', age: 42 }
  }
}
```

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

31. What is reconciliation?

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

32. How to set state with a dynamic key name?

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

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

33. What would be the common mistake of function being called every time the component renders?

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

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

Instead, pass the function itself without parenthesis:

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

34. Is lazy function supports named exports?

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

```
// MoreComponents.js
export const SomeComponent = /* ... */;
```

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

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

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

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

```
import React, { lazy } from 'react';
const SomeComponent = lazy(() => import("./IntermediateComponent.js"));
```

35. Why React uses className over class attribute?

class is a keyword in JavaScript, and JSX is an extension of JavaScript. That's the principal reason why React uses className instead of class. Pass a string as the className prop.

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

36. What are fragments?

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

```
render() {
  return (
    <React.Fragment>
      <ChildA />
      <ChildB />
      <ChildC />
    </React.Fragment>
  )
}
```

There is also a *shorter syntax*, but it's not supported in many tools:

```
render() {
  return (
```

```
    <>
      <ChildA />
      <ChildB />
      <ChildC />
    </>
  )
}
```

37. What are stateless components?

If the behaviour is independent of its state then it can be a stateless component. You can use either a function or a class for creating stateless components. But unless you need to use a lifecycle hook in your components, you should go for function components. There are a lot of benefits if you decide to use function components here; they are easy to write, understand, and test, a little faster, and you can avoid this keyword altogether.

38. What are stateful components?

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

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

  render() {
    // ...
  }
}
```

React 16.8 Update:

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

The Equivalent Functional Component

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

```
const App = (props) => {
  const [count, setCount] = useState(0);

  return (
    // JSX
  )
}
```

39. 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:

i. PropTypes.number
ii. PropTypes.string
iii. PropTypes.array
iv. PropTypes.object
v. PropTypes.func
vi. PropTypes.node
vii. PropTypes.element
viii. PropTypes.bool
ix. PropTypes.symbol
x. 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
  }
```

40. What are the advantages of React?

The list of main advantages of React,

i. Increases the application's performance with *Virtual DOM*.
ii. JSX makes code easy to read and write.
iii. It renders both on client and server side (*SSR*).

iv. Easy to integrate with frameworks (Angular, Backbone) since it is only a view library.

v. Easy to write unit and integration tests with tools such as Jest.

41. What are the limitations of React?

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

i. React is just a view library, not a full framework.

ii. There is a learning curve for beginners who are new to web development.

iii. Integrating React into a traditional MVC framework requires some additional configuration.

iv. The code complexity increases with inline templating and JSX.

v. Too many smaller components leading to over engineering or boilerplate.

42. 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:

i. render()

ii. hydrate()

iii. unmountComponentAtNode()

iv. findDOMNode()

v. createPortal()

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

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

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

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

44. 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).

45. 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>
)
```

46. 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>
```

47. How do you memoize a component?

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

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

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

const MemoizedFoo = memoize.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);
```

48. What is CRA and its benefits?

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

Let's create Todo App using *CRA*:

```
# Installation
$ npm install -g create-react-app

# Create new project
$ create-react-app todo-app
$ cd todo-app

# Build, test and run
$ npm run build
$ npm run test
$ npm start
```

It includes everything we need to build a React app:

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

49. What are the lifecycle methods deprecated in React v16?

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

i. componentWillMount()
ii. componentWillReceiveProps()
iii. componentWillUpdate()

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

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

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

```
class MyComponent extends React.Component {
  static getDerivedStateFromProps(props, state) {
    // ...
  }
}
```

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

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

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

```
class MyComponent extends React.Component {
  getSnapshotBeforeUpdate(prevProps, prevState) {
    // ...
  }
}
```

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

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

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

i. static methods
ii. constructor()

iii. getChildContext()
iv. componentWillMount()
v. componentDidMount()
vi. componentWillReceiveProps()
vii. shouldComponentUpdate()
viii. componentWillUpdate()
ix. componentDidUpdate()
x. componentWillUnmount()
xi. click handlers or event handlers like onClickSubmit() or onChangeDescription()
xii. getter methods for render like getSelectReason() or getFooterContent()
xiii. optional render methods like renderNavigation() or renderProfilePicture()
xiv. render()

53. What is strict mode in React?

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

```
import React from 'react'

function ExampleApplication() {
 return (
  <div>
   <Header />
   <React.StrictMode>
    <div>
     <ComponentOne />
     <ComponentTwo />
    </div>
   </React.StrictMode>
   <Header />
  </div>
 )
}
```

In the example above, the *strict mode* checks apply

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

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

Using super(props):

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props)
    console.log(this.props) // { name: 'John', ... }
  }
}
```

Using super():

```
class MyComponent extends React.Component {
  constructor(props) {
    super()
    console.log(this.props) // undefined
  }
}
```

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

55. 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().

56. 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'} />
```

57. 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>
```

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

The better approach is to use Array.prototype.filter() method.
For example, let's create a removeItem() method for updating the state.

```
removeItem(index) {
  this.setState({
    data: this.state.data.filter((item, i) => i !== index)
  })
}
```

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

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

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

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

```
class App extends React.Component{
  componentDidMount() {
    this.nameInput.focus()
  }

  render() {
    return (
      <div>
        <input
          defaultValue={'Won\'t focus'}
        />
        <input
          ref={(input) => this.nameInput = input}
```

```
          defaultValue={'Will focus'}
        />
      </div>
    )
  }
}

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

Also in Functional component (react 16.08 and above)

```
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'))
```

61. What are the possible ways of updating objects in state?

i. **Calling setState() with an object to merge with state:**

- Using Object.assign() to create a copy of the object:

```
const user = Object.assign({}, this.state.user, { age: 42 })
        this.setState({ user })
```

- Using *spread operator*:

```
const user = { ...this.state.user, age: 42 }
        this.setState({ user })
```

ii. **Calling setState() with a function:**

```
this.setState(prevState => ({
  user: {
    ...prevState.user,
    age: 42
  }
        }))
```

62. How to update a component every second?

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

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

componentWillUnmount() {
  clearInterval(this.interval)
}
```

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

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

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

```
class MyComponent extends React.Component {
 constructor(props) {
  super(props)
  this.state = {
   employees: [],
   error: null
  }
 }

 componentDidMount() {
  fetch('https://api.example.com/items')
   .then(res => res.json())
   .then(
    (result) => {
     this.setState({
      employees: result.employees
     })
    },
    (error) => {
     this.setState({ error })
    }
   )
 }

 render() {
  const { error, employees } = this.state
  if (error) {
   return <div>Error: {error.message}</div>;
  } else {
   return (
    <ul>
     {employees.map(employee => (
      <li key={employee.name}>
```

```
            {employee.name}-{employee.experience}
          </li>
        ))}
      </ul>
    )
  }
 }
}
```

64. 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.

65. 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')
```

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

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

At first you need to add Routes to your imports:

```
import { Routes, Router, Route } from 'react-router'
```

Then define the routes within <Switch> block:

```
<Router>
  <Routes>
    <Route {/* ... */} />
```

```
    <Route {/* ... */} />
  </Routes>
</Router>
```

67. 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.

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

A <Routes> 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

```
<Routes>
  <Route path="/" element={<Home />}/>
  <Route path="/user" element={<User />}/>
  <Route element={<NotFound />} />
</Routes>
```

69. How to get history on React Router v4?

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

i. 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 */
})
```

ii. 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)
```

iii. 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')
```

70. How to perform automatic redirect after login?

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

```
import React, { Component } from 'react'
import { Redirect } from 'react-router'

export default class LoginComponent extends Component {
  render() {
    if (this.state.isLoggedIn === true) {
      return <Redirect to="/your/redirect/page" />
    } else {
      return <div>{'Login Please'}</div>
    }
  }
}
```

71. 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.

72. What are the core principles of Redux?

Redux follows three fundamental principles:

i. **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.

ii. **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.

iii. **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.

73. 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.

74. 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
```

75. How to dispatch an action on load?

You can dispatch an action in componentDidMount() method and in render() method you can verify the data.

```
class App extends Component {
  componentDidMount() {
    this.props.fetchData()
  }
```

```
 render() {
   return this.props.isLoaded
     ? <div>{'Loaded'}</div>
     : <div>{'Not Loaded'}</div>
 }
}

const mapStateToProps = (state) => ({
 isLoaded: state.isLoaded
})

const mapDispatchToProps = { fetchData }

export default connect(mapStateToProps, mapDispatchToProps)(App)
```

76. 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)
}
```

77. 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.

78. 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.

79. 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 }
}
```

80. 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.

81. 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).

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

82. 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.

83. 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:

i. **During action creation:**

Let's take actions.js:

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

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

ii. **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
  }
}
```

84. How to structure Redux top level directories?

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

i. **Components**: Used for *dumb* components unaware of Redux.
ii. **Containers**: Used for *smart* components connected to Redux.
iii. **Actions**: Used for all action creators, where file names correspond to part of the app.
iv. **Reducers**: Used for all reducers, where files name correspond to state key.
v. **Store**: Used for store initialization.

This structure works well for small and medium size apps.

85. 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.

86. 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.

87. What are Redux selectors and why to use them?

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

For example, to get user details from the state:

```
const getUserData = state => state.user.data
```

These selectors have two main benefits,

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

88. 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)
```

89. 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
)
```

90. 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'
}
```