### 1. What is React?

### React is a front-end JavaScript library that mainly follows the component-based approach for building user interface (UI) components for a single page application. It is also used for handling the view layer in both mobile and web apps. Moreover, react plays a crucial role in developing interactive mobile and web UIs. It was created and developed by Jordan Walke; it was deployed first on the Facebook newsfeed in 2011. ReactJS released on March 2013.

React is a widely used JavaScript library that was launched in 2011. It was created by developers at Facebook, and it is primarily used for frontend development. React uses the component-based approach, which ensures to help you build components that possess high reusability. React is well known for developing and designing complex mobile user interfaces and web applications.

**2. How React works?**

Below is the sequence of steps which gives an idea about how does react work

• If the ReactComponent changes the state, the react runs the diffing algorithm to identify the changes that are made in the virtual DOM.

• The next step is reconciliation, this is used to update the DOM as per the new features.

• Now, the virtual DOM, which is lightweight in nature and is detached from the specific implementation of the browser.

• Followingly the ReactElements which are present in virtual DOM are used to build basic nodes.

• Finally, the diffing algorithm runs faster and identifies the changes. After identification, it automatically updates the DOM with the change difference.

• React creates a virtual DOM. When state changes in a component it firstly runs a "diffing" algorithm, which identifies what has changed in the virtual DOM. The second step is reconciliation, where it updates the DOM with the results of diff.

**3. What is the meaning of create-react-app in React?**

The create-react-app in [**React**](https://intellipaat.com/react-js-certification-training-course/)is a simple command-line interface (CLI) that is used in the creation of React applications, which have no build configuration.

All tools are pre-configured when using the CLI, and this allows users to focus on the code more than on dependencies to develop the application.

The following syntax is used to start a simple project in React:

Create-react-app my-app

Making use of create-react-app is advantageous in the following way:

* Support for JSX, ES6, and flow statements
* Already built and ready auto-prefixed CSS
* Fast interactive testing components
* Live development servers that help in debugging
* Scripts to handle JSS, CSS, and other files

### ****4. How can you tell React to build in the production mode?****

[**React**](https://intellipaat.com/react-js-certification-training-course/) can be coded to directly build into production by setting the process.env.NODE\_ENV variable to production.

**Note**: When React is in production, warnings and other development features are not shown.

Ordinarily, you'd utilize Webpack's DefinePlugin strategy to set NODE\_ENV to production. This will strip out things like prototype approval and additional notices. Over that, it's likewise a smart thought to minify your code in light of the fact that React utilizes Uglify's dead-code end to strip out advancement just code and remarks, which will radically diminish the measure of your package.

**5. Why React is used?**

The following reasons make one use React for building User Interfaces (UI), and they are:

• Easy to learn nature

• Simplicity

• High scalability

• Increase performance

**6. What are the features of ReactJS?**

The features of React JS are as follows:

. React improves SEO performance

React boost the performance of the SEO to higher levels as a search engine faces the problem while reading JavaScript of high-loaded applications.

. React acts as a standard for mobile app development

It provides a transition process as an ideal solution for both mobile and web applications for building rich user interfaces.

. React makes the process of writing components easier

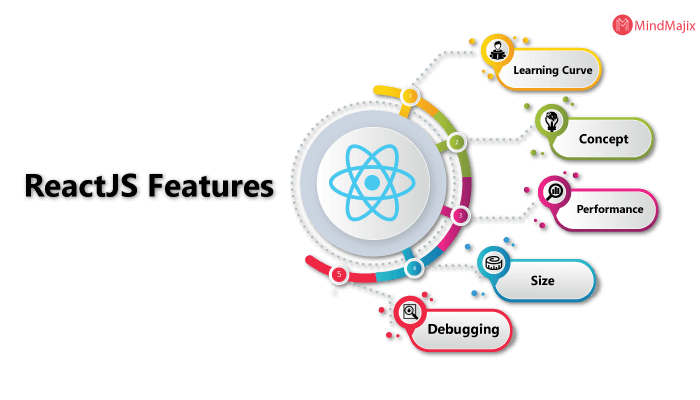
Using React along with JSX will make you write components and code efficiently and clearly.

. React increases efficiency

React boost the efficiency of components by reusing them. This is the reason why it is considered an ideal feature of React. It is considered the most reusable system component.

. React ensures stable code

It ensures the stability of the code of an application by making use of downward data flow.



ReactJS is a technology that can be trusted for complex tasks. While performing any task through it, developers need not worry about the bugs. It always ensures error-free outcomes and the best part is it offers scalable apps. It is a very fast technology and can simply be trusted for quality outcomes.

React provides users with an ample number of advantages when building an application. Some of them are as follows:

• With React, UI testing becomes very easy.

• React can integrate with Angular and other frameworks easily.

• The high readability index ensures easy understanding.

• React can be used for both client-side and server-side requirements.

• It boosts application performance and overall efficiency.

React has multiple features that are used for unique purposes. The important ones are as mentioned below:

• React makes use of a single-direction data flow model.

• It deals with complete server-side data processing and handling.

• React uses Virtual DOM that has many advantages of its own.

**7. What are the advantages of using React ?**

MVC is generally abbreviated as Model View Controller.

* Use of Virtual DOM to improve efficiency
* React uses virtual DOM to render the view. As the name suggests, virtual DOM is a virtual representation of the real DOM. Each time the data changes in a react app, a new virtual DOM gets created. Creating a virtual DOM is much faster than rendering the UI inside the browser. Therefore, with the use of virtual DOM, the efficiency of the app improves.
* Gentle learning curve
* React has a gentle learning curve when compared to frameworks like Angular. Anyone with little knowledge of javascript can start building web applications using React.
* SEO friendly
* React allows developers to develop engaging user interfaces that can be easily navigated in various search engines. It also allows server-side rendering, which boosts the SEO of an app.
* Reusable components
* React uses component-based architecture for developing applications. Components are independent and reusable bits of code. These components can be shared across various applications having similar functionality. The re-use of components increases the pace of development.
* Huge ecosystem of libraries to choose from
* React provides you the freedom to choose the tools, libraries, and architecture for developing an application based on your requirement.
* Usage of JSX makes it easier to read and write code
* Improves the performance of applications with the use of virtual DOM
* Provides an easier way to integrate with frameworks
* It can be shared and rendered both on server and client-side
* Writing integration and unit tests can be made smother by using tools

**8. Limitations of ReactJs?**

The limitations of ReactJs are:

* The code difficulty increases with inline templating and JSX.
* Too many lesser components are important to over-engineering.
* There is a knowledge curve for beginners who are novel to web development.
* React is just an outlook library, not a full-blown framework.

Yes, there are a few drawbacks which are associated with this platform. The leading drawback of ReactJS is the size of its library. It is very complex and creates a lot of confusion among the developers. Also, there are lots of developers all over the world who really don’t like the JSX and inline templating. In addition to this, there is another major limitation of ReactJS and i.e. only cover one layer of the app and i.e.View. Thus to manage the development, developers have to depend on several other technologies which consume time.

**9. Are there any disadvantages to using React?**

There are some limitations when using React as mentioned below:

* Writing code is complicated as it uses JSX and inline template formatting.
* Beginners might find it tough to cope with its syntaxes and methods.
* The library contains a huge repository of information, which might be overwhelming.
* React is a simple library and not a complete framework hence calls for dependencies.

## 10*.* Why do class methods need to be bound to a class instance?

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

class SubmitButton extends React.Component {

constructor(props) {

super(props);

this.state = {

isFormSubmitted: false

};

this.handleSubmit = this.handleSubmit.bind(this);

}

handleSubmit() {

this.setState({

isFormSubmitted: true

});

}

render() {

return (

<button onClick={this.handleSubmit}>Submit</button>

)

}

}

### ****11. What is the use of an arrow function in React?****

An arrow function is used to write an expression in React. It allows users to manually bind components easily. The functionality of arrow functions can be very useful when you are working with higher-order functions particularly.

Consider the following example:

//The usual way

render() {

return(

<MyInput onChange={this.handleChange.bind(this) } />

);

}

//Making use of the arrow function

render() {

return(

<MyInput onChange={ (e) => this.handleOnChange(e) } />

);

}

**12. Spread operator ?**

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

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

Answer

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

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

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

would be the same as:

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

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

this.setState(prevState => {

return {foo: {...prevState.foo, a: "updated"}};

});

**13. How is React different from AngularJS?**

|  |  |  |
| --- | --- | --- |
| **Comparison Factor** | **Angular** | **React** |
| Created by | Google | Facebook |
| DOM | Real DOM | Virtual DOM |
| Render Support | Client-side | Server-side |
| Architecture | Full MVC support | Only the view aspect of MVC |
| Data Binding | Unidirectional binding | Two-way binding |

The following table shows the major difference between AngularJS and React

|  |  |  |
| --- | --- | --- |
| **Factor** | **React JS** | **AngularJS** |
| **Usage of DOM** | Uses virtual DOM | Uses real DOM |
| **Language** | Uses JavaScript with the extended XML syntax | Uses TypeScript which is the superset of JavaScript |
| **App Structure** | It is represented only using the view of MVC | Made of Complete MVC |
| **Data Binding** | One-way binding | Two-way binding |

Basically, ReactJS is a limited library that builds UI parts, it is essentially not quite the same as a considerable measure of other JavaScript structures. One common example is AngularJS approaches building an app simply by expanding HTML markup and infusing different developments such as controllers at runtime. Therefore, AngularJS is exceptionally obstinate about the more noteworthy engineering of your application.

The first difference between both of them is their code dependency. ReactJS depends less on the code whereas AngularJS needs a lot of coding to be done. The packaging on React is quite strong as compared to AngularJS. Another difference is React is equipped with Virtual Dom while Angular has a Regular DOM. ReactJS is all about the components whereas AngularJS focus mainly on the Models, View as well as on Controllers. AngularJS was developed by Google while ReactJS is the outcome of Facebook. These are some of the common differences between the two.

**14. What are error boundaries?**

Introduced in the version 16 of React, Error boundaries provide a way for us to catch errors that occur in the render phase.  
  
**What is an error boundary?**  
Any component which uses one of the following lifecycle methods, is considered an error boundary.  
In what places can an error boundary detect an error?

* Render phase
* Inside a lifecycle method
* Inside the constructor

**Without using error boundaries:**

**class** CounterComponent **extends** React.Component{

constructor(props){

**super**(props);

**this**.state = {

counterValue: **0**

}

**this**.incrementCounter = **this**.incrementCounter.bind(**this**);

}

incrementCounter(){

**this**.setState(prevState => counterValue = prevState+**1**);

}

render(){

**if**(**this**.state.counter === **2**){

**throw** **new** Error('Crashed');

}

**return**(

<div>

<button onClick={**this**.incrementCounter}>Increment Value</button>

<p>Value **of** counter: {**this**.state.counterValue}</p>

</div>

)

}

}

In the code above, when the counterValue equals to 2, we throw an error inside the render method.  
When we are not using the error boundary, instead of seeing an error, we see a blank page.  
Since any error inside the render method, leads to unmounting of the component.  
To display an error that occurs inside the render method, we use error boundaries.  
  
**With error boundaries:**  
As mentioned above, error boundary is a component using one or both of the following methods:  
  
**static getDerivedStateFromError and componentDidCatch.**  
Let’s create an error boundary to handle errors in render phase:

**class** ErrorBoundary **extends** React.Component {

constructor(props) {

**super**(props);

**this**.state = { hasError: **false** };

}

static getDerivedStateFromError(error) {

**return** { hasError: **true** };

}

componentDidCatch(error, errorInfo) {

logErrorToMyService(error, errorInfo);

}

render() {

**if** (**this**.state.hasError) {

**return** <h4>Something went wrong</h4>

}

**return** **this**.props.children;

}

}

In the code above, **getDerivedStateFromError** function renders the fallback UI interface when the render method has an error.  
**componentDidCatch** logs the error information to an error tracking service.  
Now with error boundary, we can render the CounterComponent in the following way:

<ErrorBoundary>

<CounterComponent/>

</ErrorBoundary>

**15. Virtual DOM? How does react use the virtual DOM to render the UI?**

For all the available DOM objects in ReactJS, there is a parallel virtual DOM object. It is nothing but can be considered as the lighter version of the true copy and is powerful in eliminating the complex code. It is also used as a Blue Print for performing several basic experiments. Many developers also use it while practicing this technology.

A virtual DOM is a simple JavaScript object that is the exact copy of the corresponding real DOM. It can be considered as a node tree that consists of elements, their attributes, and other properties. Using the render function in React, it creates a node tree and updates it based on the changes that occur in the data model. These changes are usually triggered by users or the actions caused by the system.

As stated by the react team, virtual DOM is a concept where a virtual representation of the real DOM is kept inside the memory and is synced with the real DOM by a library such as ReactDOM.

**Why was virtual DOM introduced?** DOM manipulation is an integral part of any web application, but DOM manipulation is quite slow when compared to other operations in JavaScript.  
The efficiency of the application gets affected when several DOM manipulations are being done. Most JavaScript frameworks update the entire DOM even when a small part of the DOM changes.  
For example, consider a list that is being rendered inside the DOM. If one of the items in the list changes, the entire list gets rendered again instead of just rendering the item that was changed/updated. This is called inefficient updating.  
To address the problem of inefficient updating, the react team introduced the concept of virtual DOM.  
  
**How does it work?**

For every DOM object, there is a corresponding virtual DOM object(copy), which has the same properties.  
The main difference between the real DOM object and the virtual DOM object is that any changes in the virtual DOM object will not reflect on the screen directly. Consider a virtual DOM object as a blueprint of the real DOM object.  
Whenever a JSX element gets rendered, every virtual DOM object gets updated.  
**\*\*Note- One may think updating every virtual DOM object might be inefficient, but that’s not the case. Updating the virtual DOM is much faster than updating the real DOM since we are just updating the blueprint of the real DOM.**  
  
React uses two virtual DOMs to render the user interface. One of them is used to store the current state of the objects and the other to store the previous state of the objects.  
Whenever the virtual DOM gets updated, react compares the two virtual DOMs and gets to know about which virtual DOM objects were updated.  
After knowing which objects were updated, react renders only those objects inside the real DOM instead of rendering the complete real DOM.  
This way, with the use of virtual DOM, react solves the problem of inefficient updating.

• DOM stands for Document Object Model. It allows scripts and programs to dynamically access and updates the content, structure, and style of a document. DOM is an abstraction of a structured code called HTML, also described as HTML DOM.

• Virtual DOM is a lightweight Javascript object, which is the copy of the representation of a DOM object. It is an abstraction of HTML DOM. Virtual is quite faster compared to DOM, performs its tasks reliably.

|  |  |
| --- | --- |
| **Virtual DOM** | **Real DOM** |
| Updates faster | Updates slower |
| No memory wastage | Excess memory wastage |
| Can't update HTML directly | Update HTML directly |
| DOM manipulation is easy | DOM manipulation costly |

|  |  |
| --- | --- |
| **Virtual DOM** | **Real DOM** |
| Changes can be made easily | Changes can be expensive |
| Minimal memory wastage | High demand for memory and more wastage |
| JSX element is updated if the element exists | Creates a new DOM every time an element gets updated |
| Cannot update HTML directly | Able to directly manipulate HTML |
| Faster updates | Slow updates |

It is an in-recollection illustration of Real DOM. The depiction of a UI is kept in memory and synced with the “real” DOM. It’s a pace that happens among the render function being called and the displaying of basics on the screen.

### 16. Explain DOM diffing?

When the components are rendered twice, Virtual Dom begins checking the modifications elements have got. They represent the changed element on the page simply. There are several other elements that don’t go through changes. To cut down the changes to the DOM as an outcome of user activities, DOM doffing is considered. It is generally done to boost the performance of the browser. This is the reason for its ability to perform all the tasks quickly.

## 17*.* What is the difference between ShadowDOM and VirtualDOM?

**Virtual DOM**

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

**Shadow DOM**

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

**18. Explain Strict Mode in React.**

StrictMode is a tool added in the **version 16.3** of React to highlight potential problems in an application. It performs additional checks on the application.

function App() {

**return** (

<React.StrictMode>

<div classname="App">

<**Header**/>

<div>

Page Content

</div>

<Footer/>

</div>

</React.StrictMode>

);

}

To enable StrictMode, **<React.StrictMode>** tags need to be added inside the application:

**import** React from "react";

**import** ReactDOM from "react-dom";

**import** App from "./App";

const rootElement = document.getElementById("root");

ReactDOM.render(

<React.StrictMode>

<App />

</React.StrictMode>,

rootElement

);

StrictMode currently helps with the following issues:

* **Identifying components with unsafe lifecycle methods**  
  Certain lifecycle methods are unsafe to use in asynchronous react applications. With the use of third-party libraries it becomes difficult to ensure that certain lifecycle methods are not used.  
  StrictMode helps in providing us a warning if any of the class components uses an unsafe lifecycle method.
* **Warning about the usage of legacy string API**  
  If one is using an older version of React, **callback ref** is the recommended way to manage **refs** instead of using the **string refs**. StrictMode gives a warning if we are using **string refs** to manage refs.
* **Warning about the usage of findDOMNode**  
  Previously, findDOMNode( ) method was used to search the tree of a DOM node. This method is deprecated in React. Hence, the StrictMode gives us a warning about the usage of this method.
* Warning about the usage of legacy context API (because the API is error-prone)

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

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

The **StrictMode** component when used would benefit users immensely while creating new codebases to understand the components being used.

However, it can fit well in debugging as well because it will help solve the problem faster when it is wrapped with other components, which could be causing the problem.

Next up on these interview questions on React JS, you have to understand how to speed up rendering.

**19. What is React Fiber?**

React Fiber is a new engine in React. It is the reimplementation core algorithm in React 16.

The main goal of React Fiber is to ensure that there are incremental rendering facilities for the virtual DOM. This increases efficiency when rendering animations, gestures, etc. and also helps in assigning priority to updates based on the requirement, thereby increasing overall efficiency.

**20. JSX?**

It is basically a novel dialect of the popular JavaScript that simply integrates the HTML templates into the code of JavaScript. The browser is not capable to read the code simply and thus there is a need for this integration. Generally, WebPack or Babel tools are considered for this task. It has become a very popular approach in the present scenario among developers.

JSX stands for JavaScript XML.

It allows us to write HTML inside JavaScript and place them in the DOM without using functions like appendChild( ) or createElement( ).

As stated in the official docs of React, JSX provides syntactic sugar for React.createElement( ) function.

\*\*Note- We can create react applications without using JSX as well.

Let’s understand how JSX works:

Without using JSX, we would have to create an element by the following process:

**const** text = React.createElement('p', {}, 'This is a text');

**const** container = React.createElement('div','{}',text );

ReactDOM.render(container,rootElement);

**Using JSX,** the above code can be simplified:

const container = (

<div>

<p>This is a text</p>

</div>

);

ReactDOM.render(container,rootElement);

As one can see in the code above, we are directly using HTML inside JavaScript.

It must have one JSX element present so that the task can be accomplished easily. Having more than one expression is not an issue but probably it will slow down the process. There are also chances of confusion with more than one expression if you are new to this technology. Actually, JSX is not considered proper JavaScript. Browsers cannot read them simply. There is always a need to compile the files that contain JavaScript Code. This is usually done with the help of the JSX compiler which performs its task prior to the file entering the browser. Also, compiling is not possible in every case. It depends on a lot of factors such as the source or nature of the file or data.

When Facebook first released React to the world, they also introduced a new dialect of JavaScript called JSX that embeds raw HTML templates inside JavaScript code. JSX code by itself cannot be read by the browser; it must be transpiled into traditional JavaScript using tools like Babel and webpack. While many developers understandably have initial knee-jerk reactions against it, JSX (in tandem with ES2015) has become the defacto method of defining React components.

class MyComponent extends React.Component {

render() {

let props = this.props;

return (

<div className="my-component">

<a href={props.url}>{props.name}</a>

</div>

);

}

}

Is it possible to use the word “Class” in JSX. Why or why not?

No, it is not possible in the JSX. This is because the word “Class” is a reticent (occupied) word in JavaScript. However, you can use you are free to use the word “ClassName”. If you use the word “Class” the JSX will be translated to JavaScript immediately.

a) High-Level Component Lifecycle:

At the highest level, React components have lifecycle events that fall into 3 general classifications:

1. Initialization

2. State/Property Updates

3. Destruction

Each React component defines these events as a system for dealing with its properties, state, and rendered output. Some of these events just happen once, others happen more as often as possible; understanding these 3 general classes should help you clearly visualize when certain logic required to be applied.

For instance, a component may need to add an event audience to the DOM when it initially mounts. In any case, it ought to likely expel those event listeners when the component unmounts from the DOM with the goal that not relevant handling doesn't occur.

class MyComponent extends React.Component{

//when the component is added to the DOM...

componentDidMount(){

window.addEventListener('resize',this.onResizeHandler);

}

//when the component is removed from the DOM...

componentWillmount(){

window.addEventListener('resize',this.onResizeHandler);

}

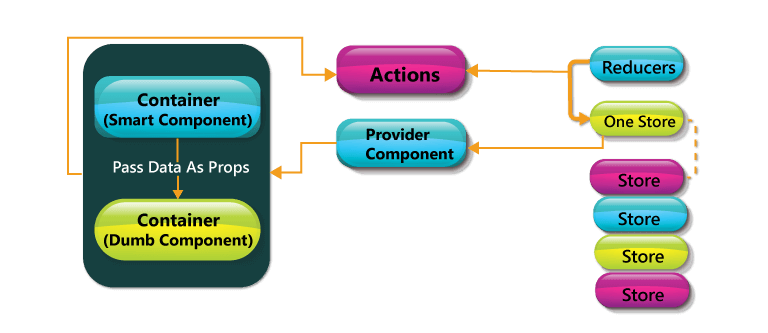
onResizeHandler(){

console.log('The window has been resized!');

}

}

b) Low-Level Component Lifecycle:



Inside these 3 general buckets exist various particular lifecycle hooks — basically unique techniques - that can be used by any React component to all the more precisely manage updates. Seeing how and when these hooks fire is vital to building stable components and will empower you to control the rendering procedure (enhancing execution).

Observe the diagram above. The events under "Initialization" just happen when a component is first initialized or added to the DOM. Thus, the events under "Devastation" just happen once (when the component is expelled from the DOM). However, the events under "Update" happen each time the properties or state of the component change.

For instance, components will naturally re-render themselves whenever their properties or states change. However, at times a component should not update - so keeping the component from re-rendering may enhance the execution of our application.

class MyComponent extends React.Component{

//only re-render if the ID has changed!

shouldComponentUpdate(nextProps, nextState;

}

}

### Is it possible to nest JSX elements into other JSX elements?

It is possible. The process is quite similar to that of nesting the HTML elements. However, there are certain things that are different in this. You must be familiar with the source and destination elements to perform this task simply.

JSX is the abbreviation of JavaScript XML. It is a file that is used in React to bring out the essence of JavaScript to React and use it for its advantages.

It even includes bringing out HTML and the easy syntax of JavaScript. This ensures that the resulting HTML file will have high readability, thereby relatively increasing the performance of the application.

Consider the following example of a JSX:

render(){

return(

<div>

<h1> Hello Intellipaat learners!</h1>

</div>

);

}

No, browsers cannot read JSX files directly. It can only read the objects provided by JavaScript. Now, to make a browser read a JSX file, it has to be transformed to a JavaScript object using JSX transformers, and only then it can be fed into the browser for further use in the pipeline.

### 21. Does React JS use HTML?

No, It uses JSX which is similar to HTML.

### ****22. How does rendering work in React?****

Rendering is an important aspect of React as every single component must be rendered. This is done using the render() function. Once the function is called, it returns an element that represents a DOM component.

It is also possible to render more than one HTML element at a time by enclosing the HTML tags and passing them through the render function.

**23. How to prevent re-renders in React?**

**Reason for re-renders in React:**  
Re-rendering of a component and it’s child components occur when props or state of the component has been changed.  
Re-rendering components that are not updated, affects the performance of an application.  
  
**How to prevent re-rendering:**  
Consider the following components:

class **Parent** extends React.Component {

state = { messageDisplayed: **false** };

componentDidMount() {

this.setState({ messageDisplayed: **true** });

}

render() {

console.log("Parent is getting rendered");

**return** (

<div className="App">

<Message />

</div>

);

}

}

class Message extends React.Component {

constructor(props) {

super(props);

this.state = { message: "Hello, this is vivek" };

}

render() {

console.log("Message is getting rendered");

**return** (

<div>

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

</div>

);

}

}

**Parent** component is the parent component and **Message** is the child component. Any change in the parent component will lead to re-rendering of the child component as well.  
To prevent the re-rendering of child component, we use the shouldComponentUpdate( ) method:  
  
**\*\*Note- Use shouldComponentUpdate( ) method only when you are sure that it’s a static component.**

class Message extends React.Component {

constructor(props) {

super(props);

this.state = { message: "Hello, this is vivek" };

}

shouldComponentUpdate() {

console.log("Does not get rendered");

**return** **false**;

}

render() {

console.log("Message is getting rendered");

**return** (

<div>

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

</div>

);

}

}

As one can see in the code above, we have returned **false** from the shouldComponentUpdate( ) method, which prevents the child component from re-rendering.

**24. Name a few techniques to optimize React app performance.**

There are many ways through which one can optimize the performance of a React app, let’s have a look at some of them:

* **Using useMemo( ) -**  
  It is a React hook that is used for caching CPU-Expensive functions.  
  Sometimes in a React app, a CPU-Expensive function gets called repeatedly due to re-renders of a component, which can lead to slow rendering.  
  useMemo( ) hook can be used to cache such functions. By using useMemo( ), the CPU-Expensive function gets called only when it is needed.
* **Using React.PureComponent -**  
  It is a base component class that checks state and props of a component to know whether the component should be updated.  
  Instead of using the simple React.Component, we can use React.PureComponent to reduce the re-renders of a component unnecessarily.
* **Maintaining State Colocation -**  
  This is a process of moving the state as close to where you need it as possible.  
  Sometimes in React app, we have a lot of unnecessary states inside the parent component which makes the code less readable and harder to maintain. Not to forget, having many states inside a single component leads to unnecessary re-renders for the component.  
  It is better to shift states which are less valuable to the parent component, to a separate component.
* **Lazy Loading -**  
  It is a technique used to reduce the load time of a React app. Lazy loading helps reduce the risk of web app performances to minimal.

**25. What would you do if your React application is rendering slowly?**

The cause of slow rendering in React is mostly because of the number of re-render operations, which are sometimes unnecessary. There are two main tools provided by React to help users here:

* **memo():** This is used to prevent all of the unnecessary re-rendering carried out by the function components.
* **PureComponent:** This is used to ensure that the unnecessary re-rendering of class components is avoided.

### 26. What is the second argument that can optionally be passed to setState and what is its purpose?

A callback work will be conjured when setState has completed and the part is re-rendered.   
Something that is not talked about a great deal is that setState is asynchronous, which is the reason it takes in a moment callback function. Ordinarily, it's best to utilize another lifecycle strategy instead of depending on this callback function, however, it's great to know it exists.

Class Training extends Course

{

this.state = {

sampleItem: 'learn',

}

handleChange = (event) => {

console.log(this.state.sampleItem)

this.setState({

sampleItem: event.target.value //event.target.value = Welcome

}, () => console.log(this.state.sampleItem))

};

**Output:**

Learn

Welcome

**27. Why is props passed to the super() function in React?**

Props gets passed onto the super() function if a user wishes to access this.props in the constructor.

Consider the following example:

class MyComponent extends React.Component {

constructor(props) {

super(props)

console.log(this.props)

// -> { icon: 'home', … }

}

}

**28. What are the predefined prop types present in React?**

There are five main predefined prop types in React. They are as follows:

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

The propTypes can be defined for the user component as shown below:

import PropTypes from 'prop-types';

class User extends React.Component {

render() {

return (

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

<h2>Age, {this.props.age}

);

}

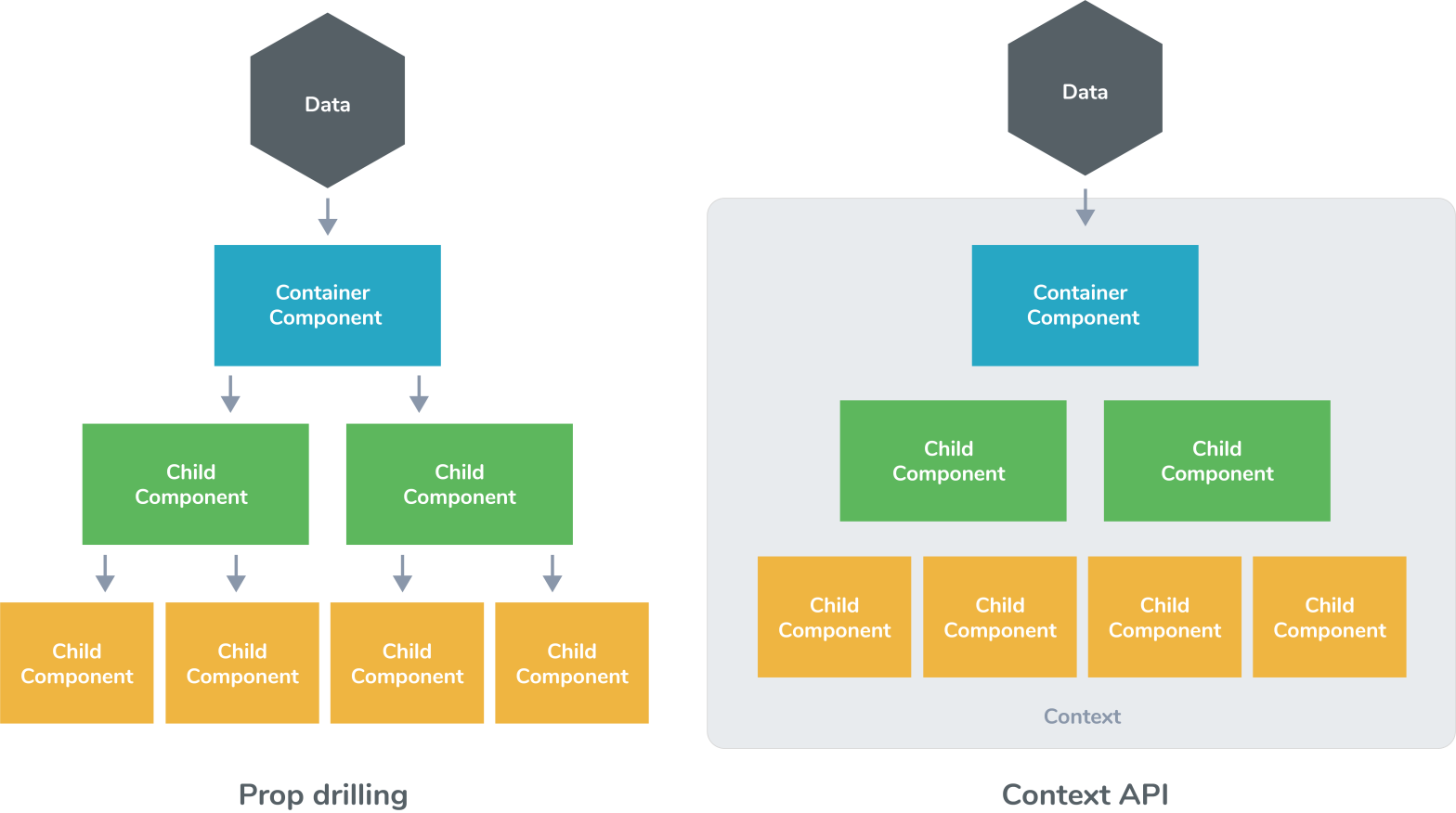
}

User.propTypes = {

name: PropTypes.string.isRequired,

age: PropTypes.number.isRequired

};

**29. What is prop drilling in React?**

Sometimes while developing React applications, there is a need to pass data from a component that is higher in the hierarchy to a component that is deeply nested.  
To pass data between such components, we pass props from a source component, and keep passing the prop to the next component in the hierarchy till we reach the deeply nested component.  
The disadvantage of using prop drilling is that the components that should otherwise be not aware of the data have access to the data.

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

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

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

**30. Why would you use force update in a React?**

In order to power a re-render if there is some form, React is not detecting that requires a revision to the UI.

**31. Explain React state and props.**

|  |  |
| --- | --- |
| **Props** | **State** |
| Immutable | Witeable/Mutable |
| Has better performance | Locally scoped |
| Can be passed to child components | can only be passed as props |
|  | has setState() method to modify properties |
|  | Changes to state can be asynchronous |
|  | Owned by its component |

**React State**  
Every component in react has a built-in **state** object, which contains all the property values that belong to that component.  
In other words, **the state object** controls the behaviour of a component. Any change in the property values of the state object leads to re-rendering of the component.

In React, State is an object that represents how the component renders and behaves. States are the sources of data and allow you to create dynamic and interactive components. They are accessed using this.state(). For changing a value in the state object, call it using this.setState() method.

**\*\*Note- State object is not available in functional components but, we can use React Hooks to add state to a functional component.**  
  
**How to declare a state object?**  
Example:

**class** Car **extends** React.Component{

constructor(props){

**super**(props);

**this**.state = {

brand: "BMW",

color: "black"

}

}

}

**How to use and update the state object?**

**class** Car **extends** React.Component {

constructor(props) {

**super**(props);

**this**.state = {

brand: "BMW",

color: "Black"

};

}

changeColor() {

**this**.setState(prevState => {

**return** { color: "Red" };

});

}

render() {

**return** (

<div>

<button onClick={() => **this**.changeColor()}>Change Color</button>

<p>{**this**.state.color}</p>

</div>

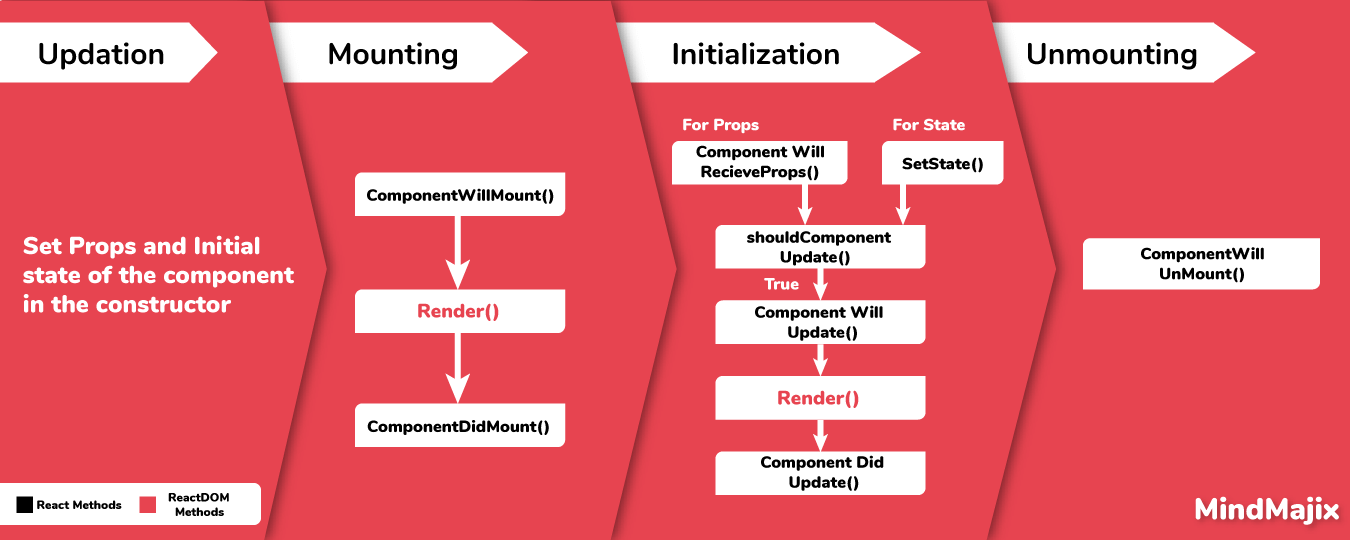
);

}

}

As one can see in the code above, we can use the state by calling **this.state.propertyName** and we can change the state object property using **setState** method.  
**React Props**

Props stand for properties in React and used for passing the information from one component to another. But the data with the Props are passed in a unidirectional flow, i.e., one way from parent to child. Further, they are read-only data, which means child components cannot change data coming from the parent.



Every react component, accepts a single object argument called **props** (which stands for “properties”).  
These props can be passed to a component using HTML attributes and the component accepts these props as an argument.  
Using props, we can pass data from one component to another.  
Passing props to a component:  
While rendering a component, we can pass the props as a HTML attribute:

<Car brand="Mercedes"/>

The component receives the props:  
In Class component:

**class** Car **extends** React.Component {

constructor(props) {

**super**(props);

**this**.state = {

brand: **this**.props.brand,

color: "Black"

};

}

}

In Functional component:

**function** Car(props) {

**let** [brand, setBrand] = useState(props.brand);

}

**\*\*\*\*Note- Props are read-only. They cannot be manipulated or changed inside a component.**

The difference between state and props are as follows:

|  |  |
| --- | --- |
| **State** | **Props** |
| The state is completely managed within a component for internal communication. | Props are directly passed to its parents with child component. |
| State can be modified using setState() method. | A particular component should never modify its own props. |
| State changes can be asynchronous | Props are read-only |

It is possible to display props on a parent component The best way to perform this task is by using the spread operator. It can also be done by listing the properties but this is a complex process.

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

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

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

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

<Element reactProp = "1" />

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

props.reactProp;

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

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

States form to be one of the vital aspects of React. It is considered as a source of data or objects that control aspects such as component behavior and rendering. In React, states are used to easily create dynamic and interactive components.

Props are the shorthand name given to properties in React. Props are read-only components that are immutable in nature. In an application, props follow a hierarchy that is passed down from parents to child components. However, the reverse is not supported. This is done to ensure that there is only a single directional flow in data at all times.

|  |  |  |
| --- | --- | --- |
| **Condition** | **Props** | **States** |
| Changes in child components | Yes | No |
| Parent component changing values | Yes | No |
| Changes inside components | No | Yes |

Both props and state are basic JavaScript objects. While both of them grasp information that influences the output of cause to be and they are diverse in their functionality relating to the constituent, i.e.,

* The state is managed within the element similar to variables confirmed within a purpose.
* Props get approved to the constituent similar to utility parameters.

**32. What do you mean by the state in ReactJs?**

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

## 33*.* Why we should not update state directly?

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

//Wrong

This.state.message =”Hello world”;

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

//Correct

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

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

**34. How to set initial state in Redux?**

You need to pass initial state as second argument to  createStore

const rootReducer = combineReducers({

  todos: todos,

  visibilityFilter: visibilityFilter

});

const initialState = {

  todos: [{id:123, name:'sudheer', completed: false}]

};

const store = createStore(

  rootReducer,

  initialState

);

**35. Redux change of state?**

For a release of action, a change in state to an application is applied; this ensures an intent to change the state will be achieved.

Example:

• The user clicks a button in the application.

• A function is called in the form of a component.

• So now an action gets dispatched by the relative container.

• This happens because the prop (which was just called in the container) is tied to an action dispatcher using mapDispatchToProps (in the container).

• Reducer on capturing the action, it intern executes a function, and this function returns a new state with specific changes.

• The state change is known by the container and modifies a specific prop in the component due to the mapStateToProps function.

**36. Explain React Hooks.**

**What are Hooks?** Hooks are functions that let us “hook into” React state and lifecycle features from **a functional component**.  
React Hooks **cannot** be used in class components. They let us write components without class.  
**Hooks** are a new addition in React 16.8. They let you use state and other React features without writing a class. With Hooks, you can extract stateful logic from a component so it can be tested independently and reused. Hooks allow you to reuse stateful logic without changing your component hierarchy. This makes it easy to share Hooks among many components or with the community.

Hooks are used to make use of the state and other features without having to explicitly write a class. Hooks were added to the React version, v16.8. The stateful logic can be extracted from a component easily, alongside testing and reusing it. All of this is done without making any changes to the component hierarchy.

## Advantages of **Hooks** :

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

Hooks don’t work inside classes (because they let you use React without classes). By using them, we can totally avoid using lifecycle methods, such as componentDidMount, componentDidUpdate, componentWillUnmount. Instead, we will use built-in hooks like useEffect .

**Why were Hooks introduced in React?**  
React hooks were introduced in the 16.8 version of React.  
Previously, functional components were called stateless components. Only class components were used for state management and lifecycle methods.  
The need to change a functional component to a class component, whenever state management or lifecycle methods were to be used, led to the development of Hooks.  
**Example of a hook:**  
**useState hook:**  
In functional components, useState hook lets us define state for a component:

**function** Person(props) {

// We are declaring a state variable called name.

// setName is a function to update/change the value of name

**let** [name, setName] = useState('');

}

The state variable “name” can be directly used inside the HTML.

use of useState(0) there:

...

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

const [moreStuff, setMoreStuff] = useState(...);

...

const setCount = () => {

setCounter(count + 1);

setMoreStuff(...);

...

};

Answer

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

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

### 37. What do you understand with the term polling?

The server needs to be monitored for updates with respect to time. The primary aim in most cases is to check whether novel comments are there or not. This process is basically considered pooling. It checks for updates approximately every 5 seconds. It is possible to change this time period easily. Pooling help keeping an eye on the users and always make sure that no negative information is present on the servers. Actually, it can create issues related to several things, and thus pooling is considered.

## 38. React Hooks: initialize state from a function

**Yes**! Consider:

const StateFromFn = () => {

const [token] = useState(() => {

let token = window.localStorage.getItem("my-token");

return token || "default#-token#"

})

return <div>Token is {token}</div>

}

## 39. React Hooks: Does React useState Hook update immediately? And how do you perform an action after useState hook has triggered?

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

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

## 40. React Hooks: What's wrong with that code?

Problem

Consider:

this.setState({

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

});

What's wrong with that code?

Answer

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

// Correct

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

counter: state.counter + props.increment

}));

**41. What is Redux form?**

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

**42. What are the main features of redux form?**

The main features of redux form are:

* Justification (sync/async) and compliance
* Field values determination via Redux store
* Formatting, normalization and parsing of field standards

Below are the major features of redux form:

1. Field values persistence via Redux store

2. Validation (sync/async) and submission

3. Formatting, parsing and normalization of field values

### ****43. What is the meaning of the component-based architecture of React?****

In [**React**](https://intellipaat.com/community/45590/how-to-learn-react-js), components are foundations used to build user interfaces for applications. With the component-based system in place, all of the individual entities become completely reusable and independent of each other. This means that rendering the application is easy and not dependent on the other components of the UI.

### 44. In ReactJS, why there is a need to capitalize on the components?

It is necessary because components are not the DOM element but they are constructors. If they are not capitalized, they can cause various issues and can confuse developers with several elements. At the same time, the problem of integration of some elements and commands can be there.

**45. Differences between controlled and uncontrolled components?**

A controlled component, as the name suggests, is a component over which React has complete control. It is the singular point of data for the forms.

An uncontrolled component is one where the form data gets handled by DOM and not the React component. This is usually done using refs in React.

There are components in the ReactJS that maintain their own internal state. They are basically considered uncontrolled components. On the other side, the components which don’t maintain any internal state are considered as controlled components in ReactJS. Controlled components can easily be controlled by several methods. Most of the React components are controlled components.

Controlled and uncontrolled components are just different approaches to handling input form elements in react.

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **Uncontrolled** | **Controlled** | **Name attrs** |
| One-time value retrieval (e.g. on submit) | ✔️ | ✔️ | ✔️ |
| Validating on submit | ✔️ | ✔️ | ✔️ |
| Field-level Validation | ❌ | ✔️ | ✔️ |
| Conditionally disabling submit button | ❌ | ✔️ | ✔️ |
| Enforcing input format | ❌ | ✔️ | ✔️ |
| several inputs for one piece of data | ❌ | ✔️ | ✔️ |
| dynamic inputs | ❌ | ✔️ | 🤔 |

**Controlled component** In a controlled component, the value of the input element is controlled by React.  
We store the state of the input element inside the code, and by using event-based callbacks, any changes made to the input element will be reflected in the code as well.  
When a user enters data inside the input element of a controlled component, onChange function gets triggered and inside the code we check whether the value entered is valid or invalid. If the value is valid, we change the state and re-render the input element with new value.  
Example of a controlled component:

function FormValidation(props) {

let [inputValue, setInputValue] = useState("");

let updateInput = e => {

setInputValue(e.target.value);

};

return (

<div>

<form>

<input type="text" value={inputValue} onChange={updateInput} />

</form>

</div>

);

}

As one can see in the code above, the value of the input element is determined by the state of the **inputValue** variable. Any changes made to the input element is handled by the **updateInput** function.  
  
**Uncontrolled component** In an uncontrolled component, the value of the input element is handled by the DOM itself.  
Input elements inside uncontrolled components work just like normal HTML input form elements.  
The state of the input element is handled by the DOM. Whenever the value of the input element is changed,event-based callbacks are not called. Basically, react does not perform any action when there are changes made to the input element.  
Whenever use enters data inside the input field, the updated data is shown directly. To access the value of the input element, we can use **ref**.  
Example of an uncontrolled component:

function FormValidation(props) {

**let** inputValue = React.createRef();

**let** handleSubmit = e => {

alert(`Input value: ${inputValue.current.value}`);

e.preventDefault();

};

**return** (

<div>

<form onSubmit={handleSubmit}>

<input **type**="text" ref={inputValue} />

<button **type**="submit">Submit</button>

</form>

</div>

);

}

As one can see in the code above, we are **not** using **onChange** function to govern the changes made to the input element. Instead, we are using **ref** to access the value of the input element.

## Controlled component and an Uncontrolled one in React:

This relates to stateful DOM components (form elements) and the React docs explain the difference:

• A Controlled Component is one that takes its current value through props and notifies changes through callbacks like onChange. A parent component "controls" it by handling the callback and managing its own state and passing the new values as props to the controlled component. You could also call this a "dumb component".

• A Uncontrolled Component is one that stores its own state internally, and you query the DOM using a ref to find its current value when you need it. This is a bit more like traditional HTML.

Most native React form components support both controlled and uncontrolled usage:

// Controlled:

<input type="text" value={value} onChange={handleChange} />

// Uncontrolled:

<input type="text" defaultValue="foo" ref={inputRef} />

// Use `inputRef.current.value` to read the current value of <input>

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

The difference between them is:

* A controlled component is a component where React is in power and is the single source of fact for the form data.
* An uncontrolled component is where your form data is handled by the DOM, in its place of within your React component.

Controlled components in React refer to the components that have the ability to maintain their state. The data is completely controlled by the parent component, and the current value is fetched by making use of props. This is done to notify about any change that occurs when using callbacks.

### 46. What’s the difference between an Element and a Component in React?

Basically, a React component describes what you need to see on the screen. Not all that basically, a React element is a protest portrayal of some UI.

A React component is a function or a class that alternatively acknowledges input and returns a React component (ordinarily by means of JSX which gets transpiled to a createElement invocation).

createElement is the thing that JSX gets transpiled to and is the thing that React uses to make React Elements (protest representations of some UI). cloneElement is utilized as a part of a request to clone a component and pass it new props. They nailed the naming on these two.

### ****47. What is the difference between cloneElement and createElement in React?****

In React, cloneElement is primarily used to clone an element and pass it to new props directly. Whereas, createElement is the entity that JSX gets compiled into. This is also used to create elements in React.

Next up on this top React interview questions and answers blog, take a look at the use of the second argument.

## 48. React.createElement?

**Question**:

const element = (

<h1 className="greeting">

Hello, world!

</h1>

);

What is equivalent of the following using React.createElement?

**Answer**:

const element = React.createElement(

'h1',

{className: 'greeting'},

'Hello, world!'

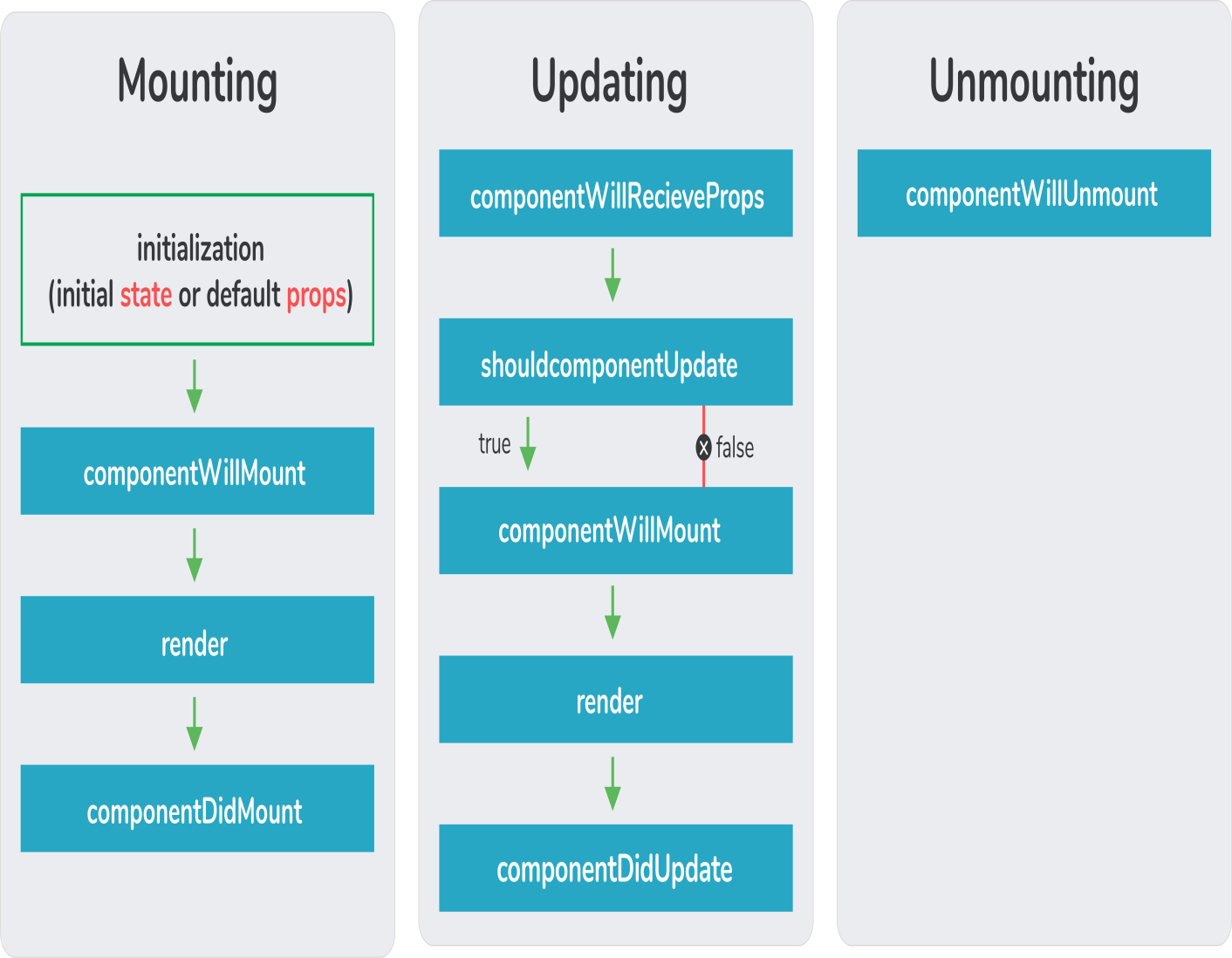
);

### ****49. What are pure components in React?****

Pure components are singular entities that are written in React. They are fast and simple to write and have the ability to replace a component that has only the render() function. This is done to ensure that the performance of the application is good and that the code is kept simple at the same time.

### 50. What are the different lifecycle methods in React?

### A standout amongst the most valuable parts of React is its segment lifecycle — so seeing precisely how segments components after some time is instrumental in building a viable application.

Every component in React has lifecycle methods that we can tap into, to trigger changes at a particular phase of the life cycle.  
Each component in react goes through three phases: **Mounting**, **Updating**, and **Unmounting**.  
There are corresponding lifecycle methods for each of the three phases:  
  
**\*\*Note- In this article, we are discussing the use of lifecycle methods in class components. For utilising lifecycle methods in functional components, react hooks are used.**  


*Mounting*:

There are four built-in lifecycle methods that are called in the **following order** when a component is mounted:  
**constructor( )** - This is called before anything else. We can set the initial state of the component inside this method. The constructor method is used to set the initial state and bind methods to the component.  
**getDerivedStateFromProps( )** - This is called before rendering the elements in the DOM.  
In this method, we can set the state of the component based on the props we received. This method is used very rarely.  
**render( )** - This is the only required method in the class component. This method returns the HTML elements which are going to be rendered inside the DOM.  
**componentDidMount( )** - It is called right after the component is rendered inside the DOM. All the statements which require the DOM nodes can be executed in this method. Network requests from a remote end-point can also be instantiated in this method.

AJAX solicitations ought to go in the componentDidMount lifecycle event. There are a couple of reasons behind this, Fiber, the following usage of React's reconciliation algorithm, will be able to begin and quit rendering as required for execution benefits. One of the exchange offs of this is componentWillMount, the other lifecycle event where it may bode well to influence an AJAX to ask for will be "non-deterministic". This means React may begin calling componentWillMount at different circumstances at whenever point it senses that it needs to. This would clearly be a bad formula for AJAX requests. You can't ensure the AJAX request won't resolve before the component mounts. In the event that it did, that would imply that you'd be attempting to setState on an unmounted component, which won't work, as well as React will holler at you for. Doing AJAX in componentDidMount will ensure that there's a component to update.

#### Updating:

Updates in react are caused by changes in state or props. Update leads to re-rendering of the component. The following methods are called when a component is re-rendered:  
**getDerivedStateFromProps( )** - This method is called again when a component is being re-rendered.  
**shouldComponentUpdate( )** - This method is called before rendering the component when new props are received. It lets React know if the component’s output is affected by the newly received props or by the state change. By default, it returns **true**.  
**render( )** - To re-render the HTML inside the DOM, the render( ) method gets called again.  
**getSnapshotBeforeUpdate( )** - This method is called just before the newly rendered HTML gets committed to the DOM. It stores the previous state of the component so that React has an idea of what parts of the DOM needs to be updated.  
**componentDidUpdate( )** - It is called after the component gets re-rendered. This method works just like the componentDidMount( ) method, the difference is that this method does not get called on initial render.

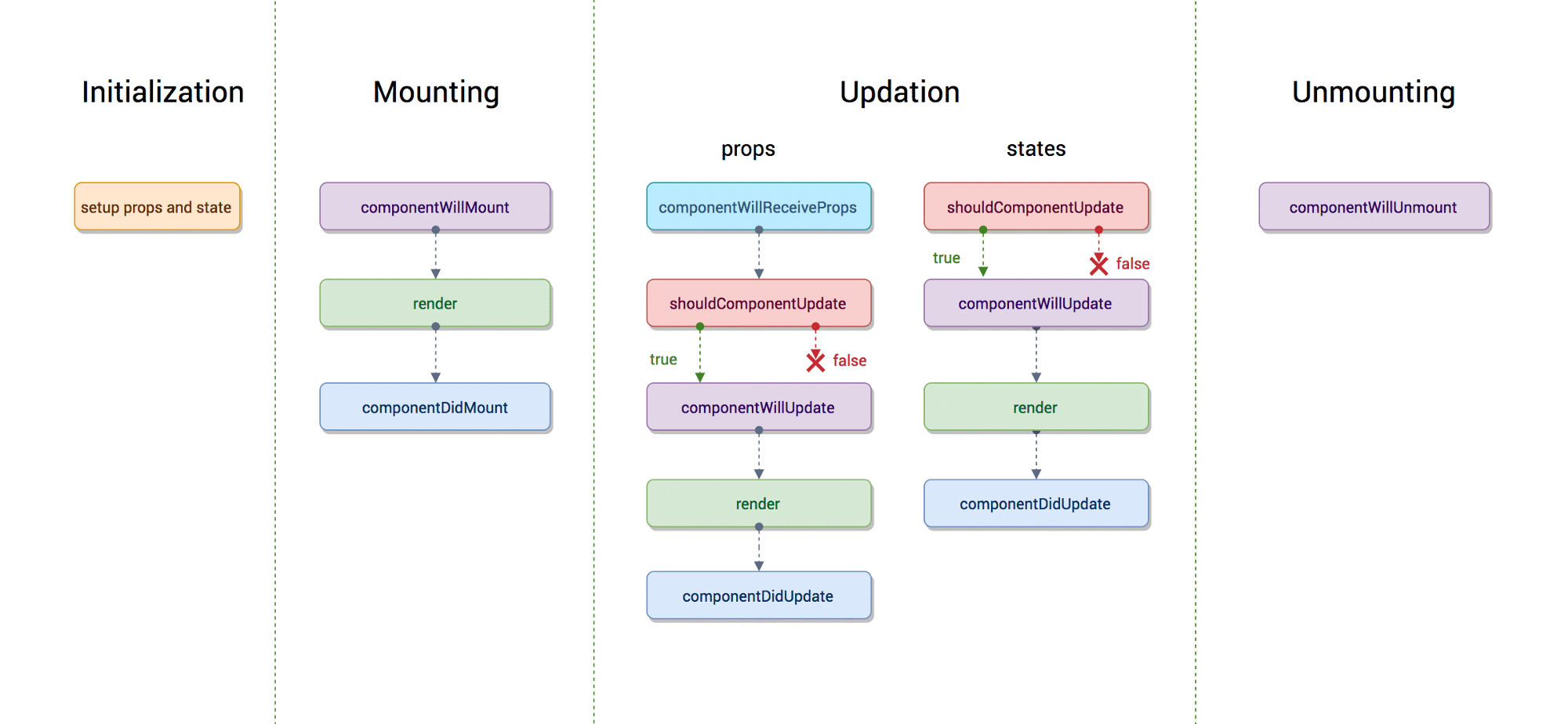
#### Unmounting:

**componentWillUnmount( )** - This method is called just before the component gets destroyed. Any clean up statements should be executed inside this method.

The **component lifecycle** is an essential part of this platform. Basically, they have lifecycle events that fall in the three prime categories which are property updates, Initialization, and third is Destruction. They are generally considered as a method of simply managing the state and properties of every reach component.

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

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



## Lifecycle methods of ReactJS :

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

**51. What are Synthetic events in React?**

React implements Synthetic events to improve the consistency and performance of applications and interfaces. The synthetic event is a cross-browser wrapper around the browser’s native event. It combines the behavior of multiple browsers into a single API to make sure events have the same properties across different browsers and platforms.

Synthetic events in React are objects that act as cross-browser wrappers, allowing for the use of native events. This is done to ensure that a variety of browsers can run the API and that the event contains all properties.

React implements Synthetic events to improve the consistency and performance of applications and interfaces. The synthetic event is a cross-browser wrapper around the browser’s native event. It combines the behavior of multiple browsers into a single API to make sure events have the same properties across different browsers and platforms.

### ****52. What are events in React?****

Whenever there are actions performed in [React](https://reactjs.org/), such as hovering the mouse or pressing a key on the keyboard, these actions trigger events. Events then perform set activities as a response to these triggers. Handling an event in React is very similar to that in the DOM architecture.

Events can be created very easily in React as shown here:

class Display extends React.Component({

show(evt) {

// Code inside

},

render() {

// Render the div with an onClick prop (value is a function)

return (

<div onClick={this.show}>Click Here</div>

);

}

});

**53. What are the different ways to style a React component?**

There are many different ways through which one can style a React component. Some of the ways are :

* **Inline Styling**  
  We can directly style an element using inline style attributes.  
  Make sure the value of style is a JavaScript object:
* class RandomComponent extends React.Component {
* render() {
* **return** (
* <div>
* <h3 style={{ color: "Yellow" }}>This is a heading</h3>
* <p style={{ fontSize: "32px" }}>This is a paragraph</p>
* </div>
* );
* }
* }
* **Using JavaScript object**  
  We can create a separate JavaScript object and set the desired style properties.  
  This object can be used as the value of the inline style attribute.
* class RandomComponent extends React.Component {
* paragraphStyles = {
* color: "Red",
* fontSize: "32px"
* };
* headingStyles = {
* color: "blue",
* fontSize: "48px"
* };
* render() {
* **return** (
* <div>
* <h3 style={this.headingStyles}>This is a heading</h3>
* <p style={this.paragraphStyles}>This is a paragraph</p>
* </div>
* );
* }
* }
* **CSS Stylesheet**  
  We can create a separate CSS file and write all the styles for the component inside that file. This file needs to be imported inside the component file.
* **import** './RandomComponent.css';
* class RandomComponent extends React.Component {
* render() {
* **return** (
* <div>
* <h3 className="heading">This is a heading</h3>
* <p className="paragraph">This is a paragraph</p>
* </div>
* );
* }
* }
* **CSS Modules**  
  We can create a separate CSS module and import this module inside our component. Create a file with “.module.css”‘ extension,  
  styles.module.css:
* **.paragraph**{
* **color**:"red";
* **border**:**1px** **solid** black;
* }

We can import this file inside the component and use it:

import styles from './styles.module.css';

class RandomComponent extends React.Component {

render() {

return (

<div>

<h3 className="heading">This is a heading</h3>

<p className={styles.paragraph} >This is a paragraph</p>

</div>

);

}

}

### 54. What do you know about React Router?

Rendering the components is an important task in ReactJS. React router is used to decide which components are to be rendered and which ones should not. It also performs dictation during several activities.

A router is very much necessary in React as it is used to manage multiple routes whenever a user types in a URL. If the route is present in the router for that corresponding URL, then the user is taken to the particular route.

To do this, the router library needs to be added in React. It can be done using the following syntax:

<switch>

<route exact path=’/’ component={Home}/>

<route path=’/posts/:id’ component={Newpost}/>

<route path=’/posts’   component={Post}/>

</switch>

**55. How is routing in React vs conventional routing?**

Differences between the conventional routing and the routing in React can be shown using the following aspects:

* **Pages**: Each view is considered as a new file in conventional routing while it is considered as a single HTML entity in React.
* **Navigation**: In conventional routing, users have to move across web pages for viewing. In React, the views are not refreshed as objects are re-issued to create new views.

### 56. What do you know about Flux?

It is an application design model used as a substitute for the more traditional MVC outline. It is not a framework but a new type of structural design that complements React and the notion of Unidirectional Data Flow.

Basically, Flux is a basic illustration that is helpful in maintaining the unidirectional data stream.  It is meant to control construed data's unique fragments to make them interface with that data without creating issues. Flux configuration is insipid; it's not specific to React applications, nor is it required to collect a React application.  Flux is basically a straightforward idea, however, you have to exhibit a profound comprehension of its usage. Applications that are built on Flux have components that can simply be tested. By simply updating the store, developers are able to manage and test any react component. It cut down the overall risk of data affection. All the applications are highly scalable and suffer no compatibility issues.

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



### 57. Compare MVC vs Flux?

MVC approaches are presently considered as outdated. Although they are capable to handle data concerns, controllers as well as UI, many developers found that it doesn’t properly work when application size increases. However, they are capable to handle some of the key issues such as eliminating the lack of data integrity as well as managing the data flow which is not properly defined. On the other side, Flux works perfectly with all the sizes irrespective of their size.

Conventional MVC designs have functioned admirably to separate the worries of data (Model), UI (View) and logic (Controller) — however many web engineers have found impediments with that approach as applications develop in measure. In particular, MVC architectures as often as possible experience 2 primary issues:

**Ineffectively defined data flow:** The cascading updates which happen crosswise over perspectives frequently prompt a tangled web of events that are hard to debug.

**Lack of data integrity:** Model data can be changed from any place, yielding erratic results over the UI.

With the Flux pattern complex, UIs never again experience the ill effects of cascading updates; any given React component will have the capacity to recreate its state in light of the information given by the store. The flux pattern likewise upholds data integrity by limiting direct access to the shared data.

While a technical interview, it is awesome to talk about the contrasts between the Flux and MVC configuration designs inside the setting of a particular illustration:

For instance, imagine we have a "master/detail" UI in which the client can choose a record from a rundown (master view) and alter it utilizing an auto-populated form (detail view).

With an MVC architecture, the data contained inside the Model is shared between both the master and detail views. Each of these perspectives may have its own particular Controller assigning updates between the Model and the View. Anytime the information contained inside the Model may be updated — and it's hard to know where precisely that change happened. Did it occur in one of the Views sharing that Model, or in one of the Controllers? Since the Model's information can be transformed by any performing artist in the application, the danger of information contamination in complex UIs is more prominent than we'd like.

With a Flux architecture, the Store data is correspondingly shared between different Views. However this data can't be straightforwardly changed — the greater part of the solicitations to update the data must go through the Action > Dispatcher chain first, eliminating the risk of arbitrary data pollution. At the point when refreshes are made to the data, it's presently significantly less demanding to find the code requesting for those progressions.

Among a plethora of advantages of using MVC in React, there are minor problems as stated below:

* A lot of memory wastage occurs.
* DOM manipulation costs a lot.
* The application becomes slow.
* Lots of dependencies are created.
* The complexity of models increases.

As far as MVC structure is concerned, the data, presentation, and logical layers are well separated and handled. Change to an application even at a smaller position may involve many changes through the application. this happens because data flow exists bidirectional as far as MVC is concerned. Maintenance of MVC structures are hardly complex, and Debugging also expects a lot of experience for it.

Flux stands closely related to redux. A story based strategy allows capturing the changes applied to the application state, the event subscription, and the current state are connected by means of components. Call back payloads are broadcasted by means of Redux.

**58. Differentiate between Flux and Redux in React?**

|  |  |  |
| --- | --- | --- |
| **Comparison Factor** | **Flux** | **Redux** |
| Components | Components connected to Flux in React | Container components directly connect |
| Dispatcher | Has a dispatcher | No dispatcher |
| Number of Stores | Single store | Multiple stores |
| State | Mutable state | Immutable state |
| Storage | Contains state and logic | State and logic are separate |

Instead of downsides, there are few compromises of using Redux over Flux that is listed below:

* **You need to learn the avoiding of mutations:** Flux is un-opinionated about mutating the data, however, Redux does not like mutations, and most of the packages which are complementary to Redux should never alter the state.
* **You have to carefully pick your packages:** While Flux principle does not try to solve the problems such as undo or redo, persistence, or the forms where Redux has extension points like store enhancers and middleware.
* **No nice Flow integration yet:** Flux allows you to do impressive static type checks that Redux does not support yet.

#### 59. What is store in redux?

Answer: The store holds the application state and supplies the helper methods for accessing the state.

Register listeners and dispatch actions. There is only one Store while using Redux. The store is configured via the createStorefunction. The single store represents the entire state.

Reducers return a state via action.

export function configureStore(initialState) {

return createStore(rootReducer, initialState);

}

The root reducer is a collection of all reducers in the application.

const rootReducer = combineReducers({

donors: donorReducer,

});

#### The Redux “store” carries together all the states, reducers, and actions that create the app. The store has multiple responsibilities:

* It holds the state of the current application from inside
* With the help of store.getState(); it allows access to the current state.
* With the help of the store.dispatch(action); it allows the state to be updated.
* With the help of the store.subscriber(listener); it allows to register listener callbacks.

### Store Methods

* getState()
* dispatch(action)
* subscribe(listener)
* replaceReducer(nextReducer)

#### Example

import { createStore } from 'redux'

const store = createStore(todos, ['Use Redux'])

function addTodo(text) {

  return {

    type: 'ADD\_TODO',

    text

  }

}

store.dispatch(addTodo('Read the docs'))

store.dispatch(addTodo('Read about the middleware'))

Yes! You require keeping your application state as miniature as possible. You don't have to drive everything in there. Only do that makes a lot of intelligence to keep something there, or it makes your life easier when using Dev Tools.

**60. How to access redux store outside a react component?**

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

store = createStore(myReducer);

export default store;

To access the redux stores outside a react component, you need to export the store from the module where it has been created with createStore.

**61. What are Higher Order Components?**

Simply put, Higher Order Component(HOC) is a function that takes in a component and returns a new component.

**When do we need a Higher Order Component?**  
While developing React applications, we might develop components that are quite similar to each other with minute differences.  
In most cases, developing similar components might not be an issue but, while developing larger applications we need to keep our code **DRY**, therefore, we want an **abstraction** that allows us to define this logic in a single place and share it across components.  
HOC allows us to create that abstraction.  
Example of a HOC:  
Consider the following components having similar functionality  
The following component displays the list of articles:

// "GlobalDataSource" is some global data source

**class** ArticlesList **extends** React.Component {

constructor(props) {

**super**(props);

**this**.handleChange = **this**.handleChange.bind(**this**);

**this**.state = {

articles: GlobalDataSource.getArticles(),

};

}

componentDidMount() {

// Listens to the changes added

GlobalDataSource.addChangeListener(**this**.handleChange);

}

componentWillUnmount() {

// Listens to the changes removed

GlobalDataSource.removeChangeListener(**this**.handleChange);

}

handleChange() {

// States gets Update whenver data source changes

**this**.setState({

articles: GlobalDataSource.getArticles(),

});

}

render() {

**return** (

<div>

{**this**.state.articles.map((article) => (

<ArticleData article={article} key={article.id} />

))}

</div>

);

}

}

The following component displays the list of users:

// "GlobalDataSource" is some global data source

**class** UsersList **extends** React.Component {

constructor(props) {

**super**(props);

**this**.handleChange = **this**.handleChange.bind(**this**);

**this**.state = {

users: GlobalDataSource.getUsers(),

};

}

componentDidMount() {

// Listens to the changes added

GlobalDataSource.addChangeListener(**this**.handleChange);

}

componentWillUnmount() {

// Listens to the changes removed

GlobalDataSource.removeChangeListener(**this**.handleChange);

}

handleChange() {

// States gets Update whenver data source changes

**this**.setState({

users: GlobalDataSource.getUsers(),

});

}

render() {

**return** (

<div>

{**this**.state.users.map((user) => (

<UserData user={user} key={user.id} />

))}

</div>

);

}

}

Notice the above components, both have similar functionality but, they are calling different methods to an API endpoint.  
Let’s create a Higher Order Component to create an abstraction:

// Higher Order Component which takes a component

// as input and returns another component

// "GlobalDataSource" is some global data source

**function** HOC(WrappedComponent, selectData) {

**return** **class** **extends** React.Component {

constructor(props) {

**super**(props);

**this**.handleChange = **this**.handleChange.bind(**this**);

**this**.state = {

data: selectData(GlobalDataSource, props),

};

}

componentDidMount() {

// Listens to the changes added

GlobalDataSource.addChangeListener(**this**.handleChange);

}

componentWillUnmount() {

// Listens to the changes removed

GlobalDataSource.removeChangeListener(**this**.handleChange);

}

handleChange() {

**this**.setState({

data: selectData(GlobalDataSource, **this**.props),

});

}

render() {

// Rendering the wrapped component with the latest data data

**return** <WrappedComponent data={**this**.state.data} {...**this**.props} />;

}

};

}

We know HOC is a function that takes in a component and returns a component.  
In the code above, we have created a function called HOC which returns a component and performs a functionality that can be shared across both **ArticlesList** component and **UsersList Component**.  
The second parameter in the HOC function is the function that calls the method on the API endpoint.  
We have reduced the duplicated code of the **componentDidUpdate** and **componentDidMount** functions.  
Using the concept of Higher Order Components, we can now render the **ArticlesList** and **UsersList** component in the following way:

**const** ArticlesListWithHOC = HOC(ArticlesList, (GlobalDataSource) => GlobalDataSource.getArticles());

**const** UsersListWithHOC = HOC(UsersList, (GlobalDataSource) => GlobalDataSource.getUsers());

Remember, we are not trying to change the functionality of each component, we are trying to share a single functionality across multiple components using HOC.

In ReactJS high order component can be defined as the function that is mainly used to collect the component and returns a new component. These components are the patterns that are extracted from React’s compositional nature. One important aspect of this component is that it is used as a reusable component logic in React. It provides us with the best way to share behavior between different React 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 as **“pure’ components”** because they can accept any dynamically provided child component but they won’t modify or copy any behavior from their input components.

const EnhancedComponent = higherOrderComponent(WrappedComponent);

HOC can be used for many use cases as below,

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

Higher-order components (HOCs) are a widely used technique in React for applying concepts that involve the component reusability logic. They are not a native part of the React API and allow users to easily reuse the code and bootstrap abstraction.

HOCs are also used to allow simple sharing of behaviors across all of the components in React, adding more advances to the efficiency and functioning of the application.

HOCs are used for a variety of tasks such as:

* Manipulation of props
* State manipulation and abstraction
* Render highjacking
* Code reusing
* Bootstrap abstraction

**62. Functional programming concepts?**

The various functional programming concepts used to structure Redux are listed below,

• Functions are treated as First-class objects.

• Capable of passing functions in the format of arguments.

• Capable of controlling flow using recursions, functions, and arrays.

• Helper functions such as reduce and map filters are used.

• Allows linking functions together.

• The state doesn’t change.

• Prioritize the order of executing the code is not really necessary.

**63. Differences between functional and class components?**

Before the introduction of Hooks in React, functional components were called stateless components and were behind class components on feature basis. After the introduction of Hooks, functional components are equivalent to class components.  
Although functional components are the new trend, the react team insists on keeping class components in React. Therefore, it is important to know how these both components differ.  
On the following basis let’s compare functional and class components:

* Decalaration  
  Functional components are nothing but JavaScript functions and therefore can be declared using an **arrow function** or the **function** keyword:
* function card(props){
* return(
* <div className="main-container">
* <h2>Title of the card</h2>
* </div>
* )
* }
* const card = (props) =>{
* return(
* <div className="main-container">
* <h2>Title of the card</h2>
* </div>
* )
* }

Class components on the other hand, are declared using the ES6 class:

class Card extends React.Component{

constructor(props){

super(props);

}

render(){

return(

<div className="main-container">

<h2>Title of the card</h2>

</div>

)

}

}

* Handling props  
  Let’s render the following component with props and analyse how functional and class components handle props:
* <StudentInfo name="Vivek" rollNumber="23" />

In functional components, the handling of props is pretty straight forward. Any prop provided as an argument to a functional component, can be directly used inside HTML elements:

function StudentInfo(props){

return(

<div className="main">

<h2>{props.name}</h2>

<h4>{props.rollNumber}</h4>

</div>

)

}

In the case of class components, props are handled in a different way:

**class** StudentInfo **extends** React.Component{

constructor(props){

**super**(props);

}

render(){

**return**(

<div className="main">

<h2>{**this**.props.name}</h2>

<h4>{**this**.props.rollNumber}</h4>

</div>

)

}

}

As we can see in the code above, **this** keyword is used in the case of class components.

* Handling state  
  Functional components use React hooks to handle state.  
  It uses the **useState** hook to set state of a variable inside the component:
* function ClassRoom(props){
* let [studentsCount,setStudentsCount] = useState(**0**);
* const addStudent = () => {
* setStudentsCount(++studentsCount);
* }
* **return**(
* <div>
* <p>Number **of** students **in** **class** room: {studentsCount}</p>
* <button onClick={addStudent}>Add Student</button>
* </div>
* )
* }

Since useState hook returns an array of two items, the first item contains the current state, and the second item is a function used to update the state.  
In the code above, using array destructuring we have set the variable name to studentsCount with a current value of “0” and setStudentsCount is the function that is used to update the state.  
For reading the state, we can see from the code above, the variable name can be directly used to read the current state of the variable.  
We cannot use React Hooks inside class components, therefore state handling is done very differently in a class component:  
Let’s take the same above example and convert it into a class component:

**class** ClassRoom **extends** React.Component{

constructor(props){

**super**(props);

**this**.state = {studentsCount : **0**};

**this**.addStudent = **this**.addStudent.bind(**this**);

}

addStudent(){

**this**.setState(**(prevState)=>**{

**return** {studentsCount: prevState.studentsCount++}

});

}

render(){

**return**(

<div>

<p>Number **of** students **in** **class** room: {**this**.state.studentsCount}</p>

<button onClick={**this**.addStudent}>Add Student</button>

</div>

)

}

}

In the code above, we see we are using **this.state** to add the variable studentsCount and setting the value to “0”.  
For reading the state, we are using **this.state.studentsCount**.  
For updating the state, we need to first bind the addStudent function to **this**. Only then, we will be able to use the **setState** function which is used to update the state.

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Class Component** | **Functional Component** |
| Syntax | This component requests you to extend from React. Component to create render function that in turn returns a react element | It is just a plain JavaScript function that accepts props as their arguments and returns the react element. |
| Life cycle hooks | Lifecycle hooks are created from the React Component. This class component makes lifecycle hooks available in it. | We cannot use lifecycle hooks in a functional component. |
| Readability | They are very difficult to test and read | They are much easier to test and read |

 Class Component Coding:

class App extends Component {

render () {

return (

<Text>Hello World!</Text>

)

}

}

Functional  Component Coding:

const PageOne = () => {

return (

<h1>Page One</h1>

);

}

If your component has a state or a lifecycle method(s), use a Class component. or else, use a Functional component.

Traditional React Components as we have seen so far are making a class with class Example extends React.Component or React.createClass(). These make stateful components on the off chance that we at any point set the state (i.e. this.setState(), getInitialState(), or this.state = {} inside a constructor()).

In the event that we have no expectation for a Component to require state, or to require lifecycle methods, we can really compose Components with a pure function, consequently the expression "**pure function Component**":

function Date(props)

{

let {msg="The date is:"} = props

let now = new Date()

return <div>

<span> {msg}</span>

<time> {now.toLocaleDateString()}</time>

</div>

}

This function that returns a React Element can be used wherever we see fit:

DOM.render(<div> <Date msg="Today is"/></div>)

You might notice that also takes a prop – we can still pass information into the Component.

Class Components:

• Class-based Components uses ES6 class syntax. It can make use of the lifecycle methods.

• Class components extend from React.Component.

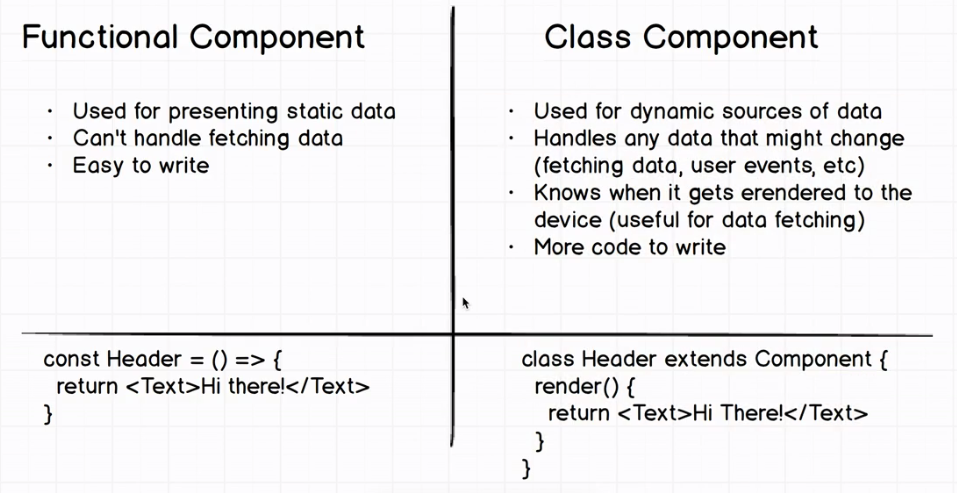
• In here you have to use this keyword to access the props and functions that you declare inside the class components.

Functional Components:

• Functional Components are simpler comparing to class-based functions.

• Functional Components mainly focuses on the UI of the application, not on the behavior.

• To be more precise these are basically render function in the class component.

• Functional Components can have state and mimic lifecycle events using Reach Hooks

**64. Can AJAX be used with React?**

Yes, any AJAX library, such as Axios and jQuery AJAX, can be used with React easily. One important thing is to maintain the states of the components, and here too, the props are passed from the parents to the child components.

Child components still cannot send back props to parents, and this factor greatly increases rendering efficiency when dynamic data is considered.

### 65. What are the stateless components?

On the off chance that Reacts components are basically state machines that produce UI markup, at that point what are stateless segments?

Stateless components (a kind of "reusable" components) are simply pure functions that render DOM construct exclusively with respect to the properties given to them.

As you can see, this component has no requirement for any internal state — not to mention a constructor or lifecycle handlers. The yield of the component is absolutely a function of the properties gave to it.

### ****66. What are stateful components in React?****

Stateful components are entities that store the changes that happen and place them into the memory. Here, the state can be changed, alongside storing information such as past, current, and future changes.

**67. How to pass data between react components?**

This task is generally performed with the help of functions. Actually, there are several functions that are provided to both parent and child components. They simply make use of them through props. Their communication should be accurate and reliable. The need for the same can be there anytime and therefore functions are considered for this task. They always make sure that information can be exchanged easily and in an efficient manner among the parent and child components.

**Parent Component to Child Component (using props)**  
With the help of props, we can send data from a parent to a child component.  
How do we do this?  
  
Consider the following Parent Component:

import ChildComponent from "./Child";

function ParentComponent(props) {

let [counter, setCounter] = useState(**0**);

let increment = () => setCounter(++counter);

**return** (

<div>

<button onClick={increment}>Increment Counter</button>

<ChildComponent counterValue={counter} />

</div>

);

}

As one can see in the code above, we are rendering the child component inside the parent component, by providing a prop called counterValue. Value of the counter is being passed from the parent to the child component.  
We can use the data passed by the parent component in the following way:

function ChildComponent(props) {

**return** (

<div>

<p>Value **of** counter: {props.counterValue}</p>

</div>

);

}

We use the **props.counterValue** to display the data passed on by the parent component.  
**Child Component to Parent Component (using callbacks)**  
This one is a bit tricky. We follow the steps below:

* Create a callback in the parent component which takes in the data needed as a parameter.
* Pass this callback as a prop to the child component.
* Send data from the child component using the callback.

We are considering the same example above but in this case, we are going to pass the updated **counterValue** from child to parent.  
**Step1 and Step2**: Create a callback in the parent component, pass this callback as a prop.

function ParentComponent(props) {

let [counter, setCounter] = useState(**0**);

let callback = valueFromChild => setCounter(valueFromChild);

**return** (

<div>

<p>Value **of** counter: {counter}</p>

<ChildComponent callbackFunc={callback} counterValue={counter} />

</div>

);

}

As one can see in the code above, we created a function called **callback** which takes in the data received from the child component as a parameter.  
Next, we passed the function **callback** as a prop to the child component.  
**Step3**: Pass data from child to the parent component.

function ChildComponent(props) {

let childCounterValue = props.counterValue;

**return** (

<div>

<button onClick={() => props.callbackFunc(++childCounterValue)}>

Increment Counter

</button>

</div>

);

}

In the code above, we have used the **props.counterValue** and set it to a variable called **childCounterValue**.  
Next, on button click, we pass the incremented childCounterValue to the **props.callbackFunc**.  
This way, we can pass data from the child to the parent component.

### 68. How to embed two components in One component?

import React from 'react';

class App extends React.Component{

render(){

return(

<div>

<Header/>

<Content/>

</div>

);

}

}

class Header extends React.Component{

render(){

return(

<div>

<h1> Header</h1>

</div>

)

}

}

class Content extends React.Component{

render(){

return(

<h2>Content</h2>

<p>The Content Text!!!</p>

</div>

)

}

}

export default App;

**69. Can you conditionally add attributes to components in React?**

Yes, there is a way in which you can add attributes to a React component when certain conditions are met.

React has the ability to omit an attribute if the value passed to it is not true.

Consider the following example:

var condition = true;

var component = (

<div

value="foo"

{ ...( condition && { disabled: true } ) } />

);

**70. Why ReactJs uses class name over class attribute?**

A class is a keyword in Javascript and JSX is an addition of Javascript. That's the major reason why React uses class Name as a replacement for class.

render() {

  return Menu

}

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

render() {

return <span className="menu navigation-menu">Menu</span>

}

## 71. How to create refs?

‘Refs’ is short for references in React. Refs are used to store a reference to a single React element or a React component. This is later returned using the render function.

They are mainly used in the following scenarios:

* To initiate imperative animations
* To join third-party DOM libraries
* To manage focus and apply media playback
* **Refs** are created using React.createRef() method and attached to React elements via the ref attribute. In order to use refs throughout the component, just assign the ref to the instance property with in constructor.

class MyComponent extends React.Component {

constructor(props) {

super(props);

this.myRef = React.createRef();

}

render() {

return <div ref={this.myRef} />;

}

}

And:

class UserForm extends Component {

handleSubmit = () => {

console.log("Input Value is: ", this.input.value)

}

render () {

return (

<form onSubmit={this.handleSubmit}>

<input

type='text'

ref={(input) => this.input = input} /> // Access DOM input in handle submit

<button type='submit'>Submit</button>

</form>

)

}

}

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

**Refs** are an escape hatch which allow you to get direct access to a DOM element or an instance of a component. In order to use them you add a ref attribute to your component whose value is a callback function which will receive the underlying DOM element or the mounted instance of the component as its first argument.

class UnControlledForm extends Component {

handleSubmit = () => {

console.log("Input Value: ", this.input.value)

}

render () {

return (

<form onSubmit={this.handleSubmit}>

<input

type='text'

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

<button type='submit'>Submit</button>

</form>

)

}

}

Above notice that our input field has a ref attribute whose value is a function. That function receives the actual DOM element of input which we then put on the instance in order to have access to it inside of the handleSubmit function.

It’s often misconstrued that you need to use a class component in order to use refs, but refs can also be used with functional components by leveraging closures in JavaScript.

function CustomForm ({handleSubmit}) {

let inputElement

return (

<form onSubmit={() => handleSubmit(inputElement.value)}>

<input

type='text'

ref={(input) => inputElement = input} />

<button type='submit'>Submit</button>

</form>

)

}

**72. What are keys in React?**

A key is a special string attribute that needs to be included when using lists of elements.  
Example of a list using key:

const ids = [**1**,**2**,**3**,**4**,**5**];

const listElements = ids.map(**(id)=>**{

**return**(

<li key={id.toString()}>

{id}

</li>

)

})

**Importance of keys**  
Keys help react identify which elements were added, changed or removed.  
Keys should be given to array elements for providing a unique identity for each element.  
Without keys, React does not understand the order or uniqueness of each element.  
With keys, React has an idea of which particular element was deleted,edited, and added.  
Keys are generally used for displaying a list of data coming from an API.  
**\*\*\*Note- Keys used within arrays should be unique among siblings. They need not be globally unique.**

Keys are used in React to check all items and to track changes actively. They are used to directly check if an item has been added or removed from a list.

Consider the following syntax:

function List ({ todos }) {

return (

<ul>

{todos.map(({ task, id} ) => <li key={id}>{task}</li>}

</ul>

)

}

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

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

**Passing props:**

class MyComponent extends React.Component {

constructor(props) {

super(props);

console.log(this.props); // Prints { name: 'sudheer',age: 30 }

}

}

**Not passing props:**

class MyComponent extends React.Component {

constructor(props) {

super();

console.log(this.props); // Prints undefined

// But Props parameter is still available

console.log(props); // Prints { name: 'sudheer',age: 30 }

}

render() {

// No difference outside constructor

console.log(this.props) // Prints { name: 'sudheer',age: 30 }

}

}

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

**74. Is there a way to avoid the requirement of binding when using React?**

Yes, there are two main ways you can use to avoid the need for binding. They are as follows:

* **Defining the Event Handler as an Inline Arrow function:**

class SubmitButton extends React.Component {

constructor(props) {

super(props);

this.state = {

isFormSubmitted: false

};

}

render() {

return (

<button onClick={() => {

this.setState({ isFormSubmitted: true });

}}>Submit</button>

)

}

}

* **Using a function component along with Hooks:**

const SubmitButton = () => {

const [isFormSubmitted, setIsFormSubmitted] = useState(false);

return (

<button onClick={() => {

setIsFormSubmitted(true);

}}>Submit</button>

)

};

Also, the Event Handler can be defined as an Arrow function, which is eventually assigned to a Class Field to obtain similar results.

**75. What is the difference between using getInitialState and constructors in React?**

When using ES6, users must initialize the state in the constructor and the getInitialState method is defined. This is done using React.createClass as shown in the below example:

class MyComponent extends React.Component {

constructor(props) {

super(props);

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

}

}

This above piece of code is equivalent to the following:

var MyComponent = React.createClass({

getInitialState() {

return { /\* initial state \*/ };

},

});

### 76. What is Redux?

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

It is one of the most in-demand libraries for front-end development in today’s growing world. It is defined as the predictable state container mainly designed for JavaScript apps and also it is used for managing the entire state of an application. Redux is very small in size and has no dependencies. It builds applications that are easy to deploy in different environments and easy to test. Redux is very small in size and has no dependencies.

A method of handling the state (or data) of an application.

Redux is used to store the state of the application in a single entity. This simple entity is usually a JavaScript object. Changing states can be done by pushing out actions from the application and writing corresponding objects for them that are used to modify the states.

For example:

{

first\_name: ‘John’,

last\_name : ‘Kelly’,

age: 25

}

All of the data is retained by Redux (also called a store).

There are many **advantages** of Redux, and some of them are as given below:

|  |  |
| --- | --- |
| Organized Approach | Redux requires code to be organized, thereby making it consistent and easy to work with |
| Testing Ability | Redux functions are small and isolated, making the code more independent and testable |
| Tools | Developers can track actions and all of the tools in React using Redux easily |
| Community | Redux has a larger community, helping users with efficient and easy-to-use libraries |

Redux is majorly used in combination with reacting. It also has the ability to get used to other view libraries too. Some of the famous entities like AngularJS, Vue.js, and Meteor. It can get combined with Redux easily. This is a key reason for the popularity of Redux in its ecosystem. So many articles, tutorials, middleware, tools, and boilerplates are available.

It is a conventional shape container for JavaScript apps based on the Flux design pattern. Redux can be used jointly with ReactJs, or with any further view library. It is small (about 2kB) and has no dependencies.

**77.Where can Redux be used?**

Redux is majorly used is a mixture with reacting. It also has the capability to get used with other view libraries too. Some of the renowned entities like AngularJS, Vue.js, and Meteor can get mutual with Redux easily.

**78. Benefits of Redux?**

• Maintainability: The maintenance of Redux becomes easier due to strict code structure and organization.

• Organization: code organization is very strict; hence the stability of the code is high, which intern increases the work to be much easier.

• Server rendering: This is useful, particularly to the preliminary render, which keeps up a better user experience or search engine optimization. The server-side created stores are forwarded to the client-side.

• Developer tools: It is Highly traceable, so changes in position and changes in the application; all such instances make the developers have a real-time experience.

• Ease of testing: The first rule of writing testable code is to write small functions that do only one thing and that are independent. Redux’s code is made of functions that used to be: small, pure, and isolated.

**79. What are the core principles of Redux?**

Redux follows three fundamental principles:

1. **Single source of truth:** The state of your whole application is stored in an object tree within a single store. The single state tree makes it easier to keep track of changes over time and debug or inspect the application.

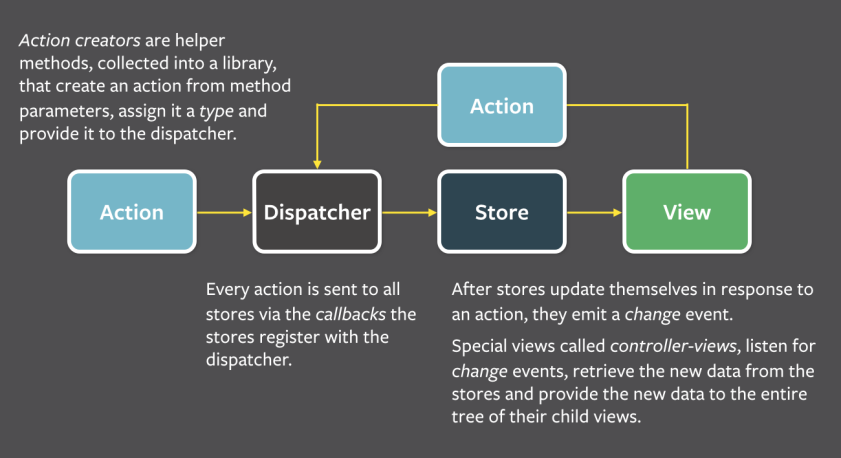
2. **State is ready only:** The only way to change the state is to emit an action, an object describing what happened. This ensures that neither the views nor the network callbacks will ever write directly to the state.

3. **Changes are made with pure functions:** To specify how the state tree is transformed by actions, you write pure reducers(Reducers are just pure functions that take the previous state and an action, and return the next state).

**80. How to structure Redux top level directories?**

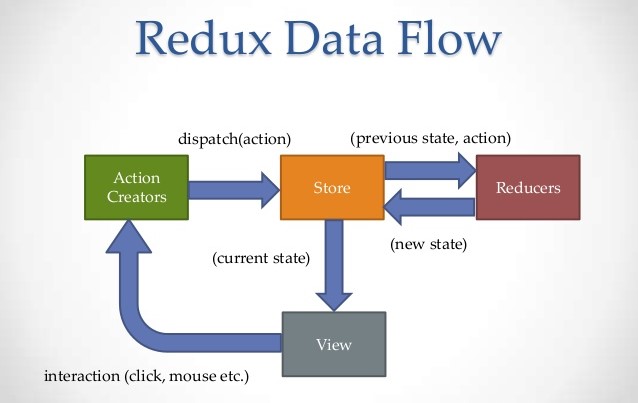
Redux consists of four main **components** as shown below:

* **Action**: An object that describes the call
* **Reducer**: The state change storage unit
* **Store**: the state and object tree storage
* **View**: Displays data provided by the store

****

All the applications have multiple top-level directories as mentioned below:

* **Components**: it is used for “dumb” React components that are unfamiliar with Redux.
* **Containers**: It is used for “smart” React components which are connected to the Redux.
* **Actions**: It is used for all the action creators, where the file name should be corresponding to the part of the app.
* **Reducers**: It is used for all the reducers where the file name is corresponding to the state key.
* **Store**: it is used for store initialization. This directory works best in small and mid-level size apps.

**

#### 81. What is the typical flow of data in a React + Redux app?

**Answer:** Call-back from UI component dispatches an act with a payload; these dispatched actions are intercepted and established by the reducers. This interception will produce a new application state. From here, the actions will be propagated down through a ladder of components from Redux store.

#### 82. Redux workflow features?

* Reset: Allow resetting the state of the store
* Revert: Rollback to the last dedicated state
* Sweep: All disabled actions that you might have fired by error will be detached
* commit: makes the present state the initial state

#### 83. Explain action’s in Redux?

**Actions in Redux are functions that return an action object. The action-type and the action data are packed in the action object. Which also allows a donor to be added to the system. Actions send data between the store and the application. The actions produce all information retrieved by the store.**

**export function addDonorAction(donor) {**

**return {**

**type: actionTypes.addDonor,**

**donor,**

**};**

**}**

Actions are the plain JavaScript objects which contain a type field. Action can be considered as an event that can describe something that has happened in the application.

Always remember actions should contain a small amount of information that is needed to mention what has happened.

#### Example

const addTodoAction = {

      type: 'ADD',

      payload: 'Buy-milk'

}

**84. What is the difference between mapStateToProps() and mapDispatchToProps()?**

| **mapStateToProps()** | **mapDispatchToProps()** |
| --- | --- |
| It is a function that is used to provide the stored data to the component. | It is a function that is used to provide the action creators with props to the component. |
| All the results of mapStateToProps() should be the plain object that will later be merged into the component’s prop. | By mapDispatchToProps(), all the action creators are wrapped in the dispatcher call so that they may be called upon directly and will be merged into the component’s prop. |
| It is used to connect the redux state to the props of the react component. | It is used to connect redux actions to the react props. |

### 85. What is meant by event handling?

To capture the user’s information and other similar data, the event handling system is considered. It is generally done through DOM elements that are present in the code. This task is simple to accomplish. Two-way communication is considered in this approach.

**86. What are reducers in redux?**

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

(previousState, action) => newState

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

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

The state of a store is updated by means of **reducer** functions. A stable collection of reducers form a store, and each of the stores maintains a separate state associated with itself. To update the array of donors, we should define a donor application.

The reducer as follows.

export default function donorReducer(state = [], action) {

switch (action.type) {

case actionTypes.addDonor:

return [...state, action.donor];

default:

return state;

}

}

The reducers receive the initial state and action. Based on the action type, it returns a new state for the store. The state maintained by reducers is immutable. The below-given reducer it holds the current state and action as an argument for it and then returns the next.

state:function handelingAuthentication(st, actn)

{

return \_.assign({}, st,

{

auth: actn.pyload

});

}

The reducers in redux are the pure functions that take the previous state and an action, and then it returns to the next state.  
**(previousState, action) => newState**

It is known as the reducer because they are the type of function that would pass to Array.prototype.reduce(reducer, ?initialValue). It is very essential to ensure that the reducer stays pure.

**To maintain this, there are few things that you should never do inside the reducer:**

* Modify its argument
* Make sure not to perform some side effects such as routing transitions and API calls
* Call non-pure functions, e.g. Date.now() or Math.random().

#### Example

const initialState = { value: 0 }

function counterReducer(state = initialState, action) {

  // Check to see if the reducer cares about this action

  if (action.type === 'counter/incremented') {

    // If so, make a copy of `state`

    return {

      ...state,

      // and update the copy with the new value

      value: state.value + 1

    }

  }

  // otherwise return the existing state unchanged

  return state

}

**87. What is redux-saga?**

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

npm install --save redux-saga

It is a documentation that aims to make elevation effects in redux applications easier and superior. It is obtainable in NPM as:

npm install --save redux-saga

**88. What are the differences between redux-saga and redux-thunk?**

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

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

**89. Define Redux Thunk?**

It is known as middleware that permits you to mark achievement creators that revisit a function in its place of an act. The Thunk can be utilized to hold up the post of an action. The internal function receives the layup methods to transmit and get state () as parameters.

**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.

### 90. What is the Use of Redux thunk?

Redux Thunk middleware is something that allows the developers to write the action creators that return functions, not actions. The [redux-thunk](https://www.npmjs.com/package/redux-thunk) can also be used for postponing the dispatch of action or to dispatch just if a specific condition is met. The inner function gets the “store” methods dispatch and getState() as the parameters.

Redux thunk acts as middleware which allows an individual to write action creators that return functions instead of actions. This is also used as a delay function in order to delay the dispatch of action if a certain condition is met. The two store methods getState() and dispatch() are provided as parameters to the inner function.

In order to activate Redux thunk, we must first use applyMiddleware() method as shown below:

import{ createStore, applyMiddleware } from 'redux';

import thunk from 'redux-thunk';

import rootReducer from './reducers/index';

//Note: this API requires redux@>=3.1.0

const store= createStore(

rootReducer,

applyMiddleware(thunk)

);

**91. 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;

So you just need to pass each piece of middleware you'd like. For example, you can add Redux Thunk and logger middlewares as an argument as below,

import { createStore, applyMiddleware } from 'redux'

const createStoreWithMiddleware = applyMiddleware(ReduxThunk, logger)(createStore);

For adding multiple middlewares to Redux, you can use applyMiddleware by which the developer can pass each piece of middleware as the new or another argument. As per your preferences, you just need to pass every single piece of middleware.

For instance, one can add the Redux Thunk and the logger middleware as the argument just as below:

#### Example

import { createStore, applyMiddleware } from 'redux'

const createStoreWithMiddleware = applyMiddleware(ReduxThunk, logger)(createStore);

**92. How to add multiple middlewares to Redux?**

You can use applyMiddleware where you can pass each piece of middleware as a new argument.

**93. How to use connect from react redux?**

You need to follow two steps to use your store in your container 1. **Use mapStateToProps():** It maps the state variables from your store to the props that you specify 2. **Connect the above props to your container:** The object returned by the mapStateToProps component is connected to the container. You can import connect from react-redux like

import React from 'react';

import { connect } from 'react-redux';

class App extends React.Component {

render() {

return <div>{this.props.containerData}</div>;

}

}

function mapStateToProps(state) {

return { containerData: state.appData };

}

export default connect(mapStateToProps)(App);

function mapStateToProps(state) {

return { containerData: state.data };

}

export default connect(mapStateToProps)(App);

**94. What is the purpose of the constants in Redux?**

When you use an IDE, the constants allow you to find all the usages of specific functionality across the project. It also prevents you from making silly mistakes which are usually caused by typos; in that case, you will receive a Reference Error immediately.

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

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

export const ADD\_TODO = 'ADD\_TODO';

export const DELETE\_TODO = 'DELETE\_TODO';

export const EDIT\_TODO = 'EDIT\_TODO';

export const COMPLETE\_TODO = 'COMPLETE\_TODO';

export const COMPLETE\_ALL = 'COMPLETE\_ALL';

export const CLEAR\_COMPLETED = 'CLEAR\_COMPLETED';

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

import { ADD\_TODO } from './actionTypes';

export function addTodo(text) {

return { type: ADD\_TODO, text };

}

1. **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

}

};

**95. Are there any similarities between Redux and RxJS?**

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

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

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

**96. What is the difference between React context and React redux?**

| **React Context** | **React Redux** |
| --- | --- |
| This can be used in the application directly and best for passing the data to the deeply nested components. | To use this in the application, you need to code it separately and then need to merge them. |
| Context API doesn’t provide a large number of features. | Redux is much more powerful and provides a large number of features |

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

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

**97. How is Relay different from Redux?**

| **Relay** | **Redux** |
| --- | --- |
| It only manages the state that originated from the server. | Redux manages and handles all the states of the application. |
| Relay caches and optimizes the data. | Redux does not handle data fetching; however, this can be done manually. |

**98. What is Redux DevTools? Also, explain the features of 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.

**99. What are the features of Redux DevTools?**

Below are the major features of Redux devTools

1. Lets you inspect every state and action payload

2. Lets you go back in time by “cancelling” actions

3. If you change the reducer code, each “staged” action will be re-evaluated

4. If the reducers throw, you will see during which action this happened, and what the error was

5. With persistState() store enhancer, you can persist debug sessions across page reloads