

# React Interview Questions

## 1. Differentiate between stateful and stateless components.

**State:** A State is an object inside the constructor method of a class which is a must in the stateful components. It is used for internal communication inside a component. It allows you to create components that are interactive and reusable. It is mutable and can only be changed by using the setState() method.

### **Stateful Components:**

Stateful components are those components which have a state. The state gets initialized in the constructor. It stores information about the component's state change in memory. It may get changed depending upon the action of the component or child components. They are also called Smart Components.

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

class StateExample extends Component {

  constructor(){

    super();

    this.state={

      first_name: 'Shruti',

      last_name: 'Priya'

    }

  }

}
```

```
render(){  
  return (  
    <div>  
  
    <p> Class Component </p>  
  
    <p>{this.state.first_name}</p>  
  
    <p>{this.state.last_name}</p>  
  
    </div>  
  
  )  
}  
}  
export default StateExample;
```

### **Stateless Components:**

Stateless components are those components which don't have any state at all, which means you can't use `this.setState` inside these components. It can be created like a normal functional component. It has no lifecycle, so it is not possible to use lifecycle methods such as `componentDidMount` and other hooks. When react renders our stateless component, all that it needs to do is just call the stateless component and pass down the props. They are also called as Dumb Components.

```
import React from 'react';  
  
function Example(props) {  
  
  return(  

```

```
<div>

  <p>{props.first_name}</p>

  <p>{props.last_name}</p>

</div>

)

}

export default Example;

*****
*
```

## 2. What is React? What are its features?

Answer:

- 1) React is a declarative, efficient, flexible open source front-end JavaScript library developed by Facebook in 2011.
- 2) It follows the component-based approach for building reusable UI components, especially for single page applications.
- 3) It is used for developing interactive view layers of web and mobile apps. It was created by Jordan Walke, a software engineer at Facebook.
- 4) It was initially deployed on Facebook's News Feed section in 2011 and later used in its products like WhatsApp & Instagram.

Features:

React Library is gaining quick popularity as the best framework among web developers. The main features of React are:

1. JSX
2. Components
3. One-way Data Binding
4. Virtual DOM
5. Client Side Rendering
6. Performance

\*\*\*\*\*  
\*

### **3. Differentiate between states and props.??**

#### State:

The state is an updatable structure that is used to contain data or information about the component and can change over time. The change in state can happen as a response to user action or system event. It is the heart of the react component which determines the behavior of the component and how it will render. A state must be kept as simple as possible. It represents the component's local state or information. It can only be accessed or modified inside the component or by the component directly.

#### Props:

Props are read-only components. It is an object which stores the value of attributes of a tag and work similar to the HTML attributes. It allows passing data from one component to other components. It is similar to function arguments and can be passed to the component the same way as arguments passed in a function. Props are immutable so we cannot modify the props from inside the component.

\*\*\*\*\*  
\*

#### **4. What are the different phases of React component's lifecycle?**

In ReactJS, every component creation process involves various lifecycle methods. These lifecycle methods are termed as component's lifecycle. These lifecycle methods are called at various points during a component's life. The lifecycle of the component is divided into four phases. They are:

1. Initial Phase
2. Mounting Phase
3. Updating Phase
4. Unmounting Phase

Each phase contains some lifecycle methods that are specific to the particular phase

\*\*\*\*\*  
\*

#### **5.What are the three principles that Redux follows?**

The three principles that redux follows are:

1. Single source of truth: The State of your entire application is stored in an object/state tree inside a single Store. The single State tree makes it easier to keep changes over time. It also makes it easier to debug or inspect the application.
2. The State is read-only: There is only one way to change the State is to emit an action, an object describing what happened. This principle ensures that neither the views nor the network callbacks can write directly to the State.

3. Changes are made with pure functions: To specify how actions transform the state tree, you need to write reducers (pure functions). Pure functions take the previous State and Action as a parameter and return a new State.

\*\*\*\*\*

\*

## 6.What is React Router?

Answer: React Router is a standard routing library system built on top of the React. It is used to create Routing in the React application using react-router package. It helps you to define multiple routes in the app. It maintains the standard structure and behavior of the application and is mainly used for developing single page web applications.

\*\*\*\*\*

\*

## 7.What is the difference between mapStateToProps() and mapDispatchToProps()?

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

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

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

```
const mapDispatchToProps = (dispatch) => {
  return {
    onClick: (id) => {
      dispatch(toggleTodo(id))
    }
  }
}
```

It is recommended to always use the “object shorthand” form for the mapDispatchToProps.

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

```
const mapDispatchToProps = ({
  onClick
})
```

\*\*\*\*\*  
\*

## 8.What do you know about controlled and uncontrolled components?

**Controlled Components:** In React, Controlled Components are those in which form's data is handled by the component's state. It takes its current value through props and makes changes through callbacks like onClick, onChange, etc. A parent

component manages its own state and passes the new values as props to the controlled component.

**Uncontrolled Components:** Uncontrolled Components are the components that are not controlled by the React state and are handled by the DOM (Document Object Model). So in order to access any value that has been entered we take the help of refs.

For instance, if we want to add a file as an input, this cannot be controlled as this depends on the browser so this is an example of an uncontrolled input.

\*\*\*\*\*  
\*

## 9. Explain Diffing Algorithm.

React uses a heuristic algorithm called the Diffing algorithm for reconciliation based on these assumptions:

Elements of different types will produce different trees

We can set which elements are static and do not need to be checked.



# Diff Algorithm



React checks the root elements for changes and the updates depend on the types of the root elements. It basically compares both the virtual doms and then updates only the node where the change has happened.

1. Element in different types: Whenever the type of the element changes in the root, react will scrap the old tree and build a new one i.e a full rebuild of the tree.
2. Elements of the same type: When the type of changed element is the same, React then checks for attributes of both versions and then only updates the node which has changed without any changes in the tree. The component will be updated in the next lifecycle call.

\*\*\*\*\*  
\*

## 10. Explain Virtual DOM?

Virtual DOM is a lightweight JavaScript object which is an in-memory representation of real DOM.

It is an intermediary step between the render function being called and the displaying of elements on the screen.

It is similar to a node tree which lists the elements, their attributes, and content as objects and their properties. The render function creates a node tree of the React components and then updates this node tree.

DOM is a document object model, created by converting HTML CSS and JS Real DOM, which is an object which gets created whenever any React application gets loaded on the screen for the first time.

Whenever React components get mounted on the screen for the first time. Now when any user makes any changes on the screen like button click because of which the state variable will get updated so in this case the changes will not directly go to Real DOM , instead in react we have a concept known as Virtual DOM.

So we are having two virtual doms, one virtual dom gets created at the time of mounting of react component so it is a copy of your real dom. Another virtual dom is the dom which contains the new changes, updated state variables values.

Now these two virtual doms will get compared with each other and will check for the new changes. This complete procedure is known as the diffing algorithm. Now the new changes will be updated in your Real dom. This procedure is known as Reconciliation.

\*\*\*\*\*  
\*

## 11. Explain Reconciliation ?

1. On the first run, both virtual DOM and real DOM tree are created
2. React works on observable patterns, hence, whenever there is a change in the state, it updates the nodes in the virtual DOM
3. Then, react compares virtual DOM with the real DOM and updates the changes. This process is called reconciliation.

\*\*\*\*\*

\*

## 12.Explain Browser Route, Route, Routes ?

**BrowserRouter:** BrowserRouter is a router implementation that uses the HTML5 history API. This module gets imported from react-router-dom. Our entire react application should be wrapped with this module so that our entire application can access routes created using the browser router.

**Routes:** This module gets imported from react-router-dom. Whatever routes our entire application is supporting should be wrapped in this module.

**Route:** Route is the conditionally show component that renders some UI when its path matches the current URL. This module defines the path and the component should be rendered respective to that route.

\*\*\*\*\*

\*

## 13.What is server side rendering ?

1. Server-side rendering (SSR) is an application's ability to convert HTML files on the server into a fully rendered HTML page for the client.
2. The web browser submits a request for information from the server, which instantly responds by sending a fully rendered page to the client. Search engines can crawl and index content prior to delivery, which is beneficial for Search Engine Optimization purposes.
3. Popular examples of server-side rendering JavaScript frameworks include: Angular server side rendering, ejs server side rendering, server side rendering Express, Gatsby server side rendering, Google server side rendering, NestJS server side rendering, Next server side

rendering, Nuxt server side rendering, React server side rendering, and Vue server side rendering.

\*\*\*\*\*

\*

## 14.What is the difference between class and functional components?

Functional Components are components with a basic syntax of JavaScript functions that return the JSX code or the HTML code.

They were earlier called as stateless components before the introduction of Hooks, because they used to accept the data as props and display it accordingly and were mainly responsible for the rendering of the UI.

```
import React from 'react';
```

```
const App=()=>{  
  return (  
    <div>  
      <h1>Prepbytes</h1>  
    </div>  
  )  
}
```

```
export default App
```

Class based or stateful Components are the ES6 classes that extend the component class of the react library. They are known as stateful components because they are responsible for the implementation of the logic. It has different phases of the React lifecycle including rendering, mounting, updating and unmounting stages.

```
import React from 'react';
```

```
class App extends React.Component{  
  render() {  
    return (  

```

```
    <div>
      <h1>Prepbytes</h1>
    </div>
  )
}
export default App
```

```
*****
*****
```

# Implementation

## 1.Differentiate between states and props.Explain with example?

Props are a JavaScript object that React components receive as an arbitrary input to produce a React element. They provide a data flow between the components. To pass the data (props) from one component to another as a parameter:

For a class component you need to define the custom HTML attributes to which you assign your data and then pass it with special React JSX syntax:

To receive props class components need to use the JavaScript keyword `this`.

```
class App extends Component {  
  render() {  
    const greeting = 'Welcome to React';  
  
    return (  
      <div>  
        <Greeting greeting={greeting} />  
      </div>  
    );  
  }  
}  
  
class Greeting extends Component {  
  render() {  
    return <h1>{this.props.greeting}</h1>;  
  }  
}  
  
export default App;
```

For a functional component props are passed as an argument to a function:

```
import React,{ Component } from 'react'

class App extends Component {
  render() {
    const greeting = " Welcome to React ";

    return (
      <div>
        <Greeting greeting={greeting} />
      </div>
    );
  }
}

const Greeting = (props) => <h1> {props.greeting} </h1>;

export default App;
```

**Output:**



# Welcome to React

State is a JavaScript object which contains data that holds the data of the components. The second part is what makes the state different compared to props. Every user interaction with your app may lead to changes in the underlying state and in the whole UI as a result. State changes over the lifetime of a React component. Examples of state:

For a class component you need to define the constructor inside the React component:



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

class Button extends Component {
  constructor(props) {
    super(props);
    this.state = { counter: 1 };
  }
  render() {
    return (
      <button>{this.state.counter}</button>
    );
  }
}
export default Button;
```

For a functional component you need to use [useState Hook](#):

```
import React from 'react';

function Counter() {
  const [count, setCount] = React.useState(1);

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

export default Counter;
```

\*\*\*\*\*  
\*\*\*\*\*

## 2. Explain React lifecycle?

In ReactJS, every component creation process involves various lifecycle methods. These lifecycle methods are termed as component's lifecycle. These lifecycle methods are not very complicated and called at various points during a component's life. The lifecycle of the component is divided into four phases. They are:

1. Initial Phase
2. Mounting Phase
3. Updating Phase
4. Unmounting Phase

Each phase contains some lifecycle methods that are specific to the particular phase. Let us discuss each of these phases one by one.

### 1. Initial Phase

It is the birth phase of a component. Here, the component starts its journey on a way to the DOM. In this phase, a component contains the default Props and initial State. These default properties are done in the constructor of a component. The initial phase only occurs once and consists of the following methods.

- `getDefaultProps()`  
It is used to specify the default value of `this.props`. It is invoked before the creation of the component or any props from the parent is passed into it.
- `getInitialState()`  
It is used to specify the default value of `this.state`. It is invoked before the creation of the component.

## 2. Mounting Phase

In this phase, the instance of a component is created and inserted into the DOM. It consists of the following methods.

- `componentWillMount()`  
This is invoked immediately before a component gets rendered into the DOM. In this case, when you call `setState()` inside this method, the component will not re-render.
- `componentDidMount()`  
This is invoked immediately after a component gets rendered and placed on the DOM. Now, you can do any DOM querying operations.
- `render()`  
This method is defined in each and every component. It is responsible for returning a single root HTML node element. If you don't want to render anything, you can return a null or false value.

## 3. Updating Phase

It is the next phase of the lifecycle of a react component. Here, we get new Props and change State. This phase also allows to handle user interaction and provide communication with the components hierarchy. The main aim of this phase is to ensure that the component is displaying the latest version of itself. Unlike the Birth or Death phase, this phase repeats again and again. This phase consists of the following methods.

- `shouldComponentUpdate()`  
It is invoked when a component decides any changes/updation to the DOM. It allows you to control the component's behavior of updating itself. If this method returns true, the component will update. Otherwise, the component will skip the updating.
- `render()`  
It is invoked to examine `this.props` and `this.state` and return one of the following types: React elements, Arrays and fragments, Booleans or null, String and Number. If `shouldComponentUpdate()` returns false, the code inside `render()` will be invoked again to ensure that the component displays itself properly.
- `componentDidUpdate()`  
It is invoked immediately after the component updating occurs. In this

method, you can put any code inside this which you want to execute once the updating occurs. This method is not invoked for the initial render.

#### 4. Unmounting Phase

It is the final phase of the react component lifecycle. It is called when a component instance is destroyed and unmounted from the DOM. This phase contains only one method and is given below.

- `componentWillUnmount()`  
This method is invoked immediately before a component is destroyed and unmounted permanently. It performs any necessary cleanup related task such as invalidating timers, event listeners, canceling network requests, or cleaning up DOM elements. If a component instance is unmounted, you cannot mount it again.

\*\*\*\*\*  
\*\*\*\*\*

### 3. Implementation of HOC ?

HOC stands for Higher Order Component. It will always be a functional component. It takes a component as input parameter and returns a new component. Let's look at an example of the most simple HOC possible

```
// Take in a component as argument WrappedComponent
function simpleHOC(WrappedComponent) {
  // And return a new anonymous component
  return class extends React.Component {
    render() {
      return <WrappedComponent {...this.props} />
    }
  }
}
```

This HOC takes a React component, `WrappedComponent`, as a parameter. It returns a new React component. The returned component contains the `WrappedComponent` as a child.

We use the HOC to create a new component like this:

```
// Create a new component
const NewComponent = simpleHOC>Hello)

// NewComponent can be used exactly like any component
// In this case, NewComponent is functionally the same as Hello
;<NewComponent />
```

**The idea with HOC is to enhance components with functions or data.**

Let's look at an example of an HOC that enhances the component.

The way we are going to do that is to add a prop to the component. Let's add a name prop with the value "React". It would look something like this:

```
// Take in a component as argument WrappedComponent
function withNameReact(WrappedComponent) {
  // And return a new anonymous component
  return class extends React.Component {
    render() {
      return <WrappedComponent name="React" {...this.props} />
    }
  }
}
```

Let's take a look at how we can use it.

First, we define the component to send in, which uses the name prop...

```
const Hello = ({ name }) => <h1>Hello {name}!</h1>
```

...then we will enhance it with our HOC.

```
const HelloReact = withNameReact(Hello)

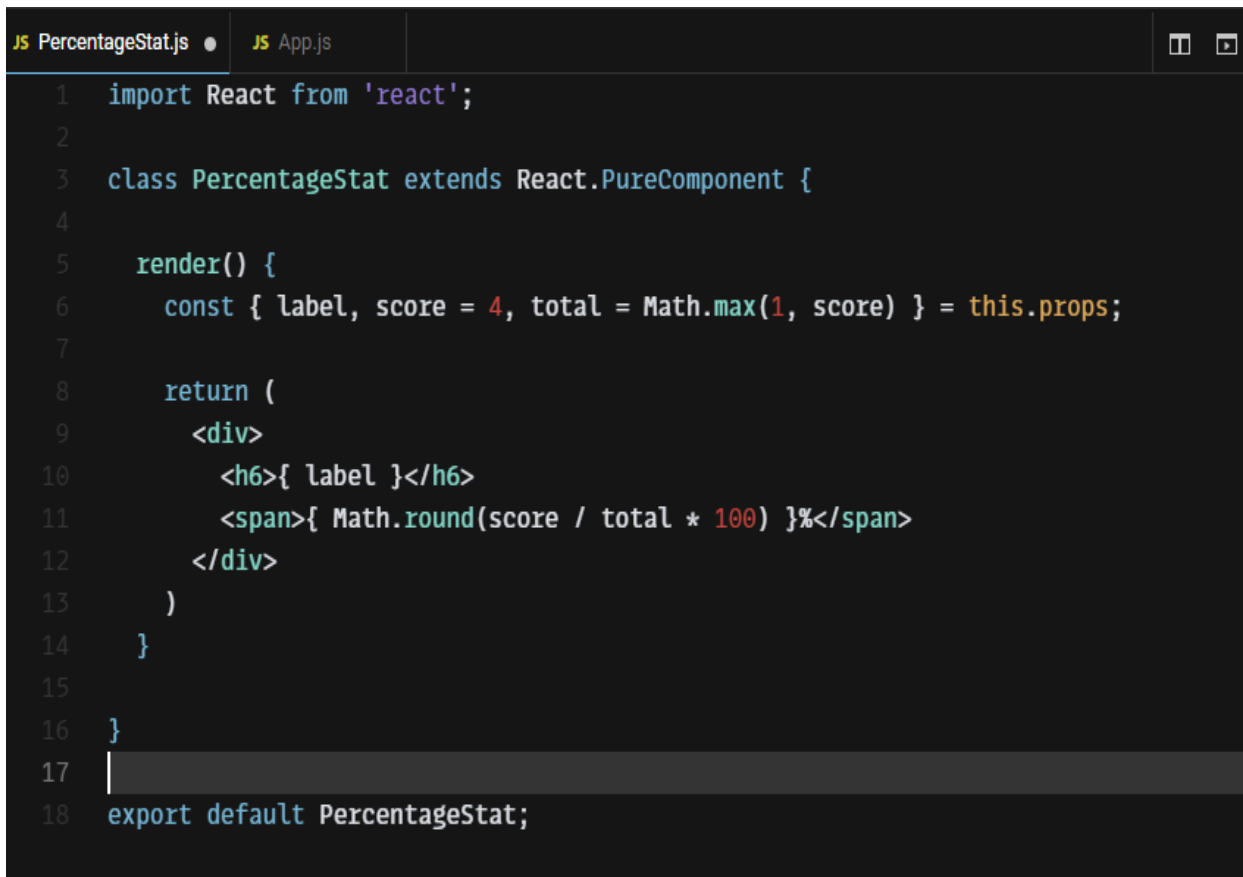
// No need to send in the name prop, it is already sent in
// by the HOC. It will output Hello React!
<HelloReact />
```

\*\*\*\*\*  
\*

## 4.Implementation of Pure Components ?

A React component is considered pure if it renders the same output for the same state and props. For this type of class component, React provides the `PureComponent` base class. Class components that extend the `React.PureComponent` class are treated as pure components.

Pure components have some performance improvements and render optimizations since React implements the `shouldComponentUpdate()` method for them with a shallow comparison for props and state.



```
1  import React from 'react';
2
3  class PercentageStat extends React.PureComponent {
4
5    render() {
6      const { label, score = 4, total = Math.max(1, score) } = this.props;
7
8      return (
9        <div>
10         <h6>{ label }</h6>
11         <span>{ Math.round(score / total * 100) }%</span>
12       </div>
13     )
14   }
15
16 }
17
18 export default PercentageStat;
```

## How to use `React.memo()`

With `React.memo()`, you can create memoized functional components that bail out of rendering on unnecessary updates using shallow comparison of props.

Using the `React.memo()` API, the previous functional component can be wrapped as follows:

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

function PercentageStat({ label, score = 0, total = Math.max(1, score) }) {
  return (
    <div>
      <h6>{ label }</h6>
      <span>{ Math.round(score / total * 100) }%</span>
    </div>
  )
}

// Wrap component using `React.memo()`
export default memo(PercentageStat);
```

\*\*\*\*\*  
\*\*\*\*\*

## 5. Explain core React Hooks- `useState` `useEffect` `useContext` ?

### `useState`:

`useState` allows you to create a state variable in a functional component. It can be imported from the `react` library.



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

function demo() {
  const [isVisible, setIsVisible] = useState(true);

  return <>{isVisible && <h1>I'm visible</h1>}</>;
}

export default demo;
```

We can use useState hook only in the functional components. The argument (initial value) can be anything, such as numbers, boolean values, etc. In this case, true (boolean). Doing this gives us two things in an array, the first is the actual state variable itself and then a function to update the state value. In this case, we're destructuring the two values right away which is the convention.

```
setIsVisible(false);
```

Now if you want to update the state variable value, you will have to call this function defined in useState hook, and it will re-render the entire component and the changes will be visible on the DOM. That's it.

## useEffect

useEffect is basically used to implement lifecycle methods in the functional component which we can not implement. So it is a replacement of lifecycle methods in functional components. Lifecycle methods like - componentDidMount, componentDidUpdate and componentWillUnmount.

This hook takes in two arguments, first is the callback function where the functionality required to be executed will be written, second is the

dependency array in which we can pass the state variables. Any change in the value of the state variable present in the dependency array will run the callback function again.

**Case 1: componentDidMount** - means function will be executed only at the time of mounting phase

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

function demo() {
  useEffect(() => {
    console.log("Like my post!");
  }, []);
}

export default demo;
```

**Case 2: componentDidMount and componentDidUpdate**- means function will be executed at the time of mounting phase and updating phase of the state variable we will pass in the dependency array.

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

function TestUseEffect () {

  const [a , setA] = useState("ABC");
  const [b, setB] = useState("ZYX");

  useEffect(()=>{
    console.log("Hello Use effect hook")
  }, [a])
}
```

### Case 3: `componentWillUnmount` - means function will be executed at the unmounting phase of component

`useEffect` contains one function named as clean up function, which gets executed at the time of updating and unmounting phase of component. In the below case since the dependency array is empty so cleanup will not be executed in updating phase, it will only be executed in the unmounting phase.

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

function TestUseEffect () {

  const [a , setA] = useState("ABC");
  const [b, setB] = useState("ZYX");

  // componentWillMount, : cleanup function in useEffect
  useEffect(()=>{
    return function cleanup(){
      console.log("Cleanup function ");
    }
  }, [])
```

\*\*\*\*\*  
\*\*

## 6. How are forms created in React?

Forms allow the users to interact with the application as well as gather information from the users. Forms can perform many tasks such as user

authentication, adding user, searching, filtering, etc. A form can contain text fields, buttons, checkbox, radio button, etc.

React offers a stateful, reactive approach to build a form. The forms in React are similar to HTML forms. But in React, the state property of the component is only updated via `setState()`, and a diff function handles their submission. This function has full access to the data which is entered by the user into a form.

You can create the state tree object like the image below and then can use one single method to handle the change in input box.

```
import React from "react";

class Form extends React.Component {
  constructor(){
    super();
    this.state={
      name:"",
      contact: null,
      user:[]
    }
  }

  manageChange = (e) => {
    this.setState({[e.target.name]:e.target.value})
  }
}
```

You can create the HTML form like below image, can use form tag and input tag to create the form and submit button to submit the form.

```

<form>
  <label htmlFor="name">Name</label>
  <input type="text" name="name" onChange={(e)=>this.manageChange(e)} id="name" value={this.state.name} />

  <label htmlFor="contact">Contact</label>
  <input type="number" name="contact" onChange={(e)=>this.manageChange(e)} id="contact" value={this.state.contact} />

  <button onClick={(event)=>this.manageSubmission(event)}>Submit</button>
</form>

```

To store the entire data given by the user in an array of objects you will have to initially create a temporary object and set the data into that and it should be used and should be assigned to the actual state variable.

```

manageSubmission = (e) => {
  e.preventDefault()
  const tempObj= {
    name:this.state.name,
    contact:this.state.contact
  }
  let tempArr=this.state.user;
  tempArr.push(tempObj);
  this.setState({user:tempArr})
}

```

```

*****
*

```

**7.What do you understand by refs in React?**

Refs are a function provided by React to access the DOM element and the React element that you might have created on your own. They are used in cases where we want to change the value of a child component, without making use of props and all. They also provide us with good functionality as we can use callbacks with them.

```
import React, { useState, useRef } from "react";
function UseRefHook () {
  const [a, setA] = useState();
  const aRef = useRef()
  return(
    <div>
      <h1>This is the explanation of useRef Hook</h1>
      <h2>{a}</h2>
      <input type="text" ref={aRef} />
      <button onClick={()=>setA(aRef.current.value)}>SetValue</button>
    </div>
  )
}
```

This ref variable that we are creating will contain an object which will have property current using which you can access the value. Also it will allow you to access all the functions of DOM using which you can modify the properties of DOM.

\*\*\*\*\*  
\*

## 8.Create routes using Router ??

<BrowserRouter>

```
<Routes>
  <Route path="/" element={<Home />}>
  <Route path="/blogs" element={<Blogs />} />
  <Route path="/contact" element={<Contact />} />
  <Route path="*" element={<NoPage />} />
</Route>
</Routes>
</BrowserRouter>
```

localhost:3000

[Home](#)  
[Blogs](#)  
[Contact](#)

---

# Home

\*\*\*\*\*  
\*\*\*\*\*

## Implementation - Difficult

### 1. Show how the data flows through Redux?

Data flow in a React-Redux application

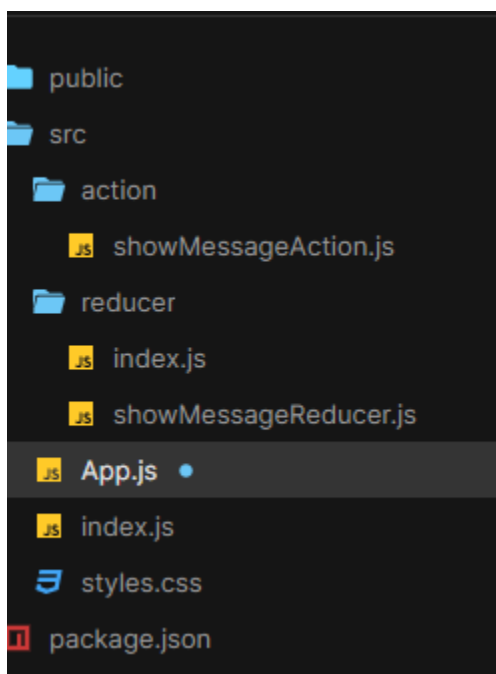
There are four fundamental concepts that govern the flow of data in React-Redux applications.

1. **Redux store:** The Redux store, simply put, is an object that holds the application state. A redux store can consist of small state objects which are combined into one large object. Any component in the application can easily access this state (store) by hooking up to it through the connect method in class components and hooks in functional components.
2. **Action creators:** Action creators, as the name suggests, are functions that return actions (objects). Action creators are invoked when the user interacts with the application through its UI (button click, form submission, etc) or at certain points in a component's lifecycle (component mounts, component unmounts, etc).
3. **Actions:** Actions are simple objects which conventionally have two properties- type(must) and payload(optional). The type property is usually a string that identifies the action, and the payload is an optional property that contains some data that is required to perform any particular task. The main function of action is to send data from the application or react components to the Redux store.
4. **Reducers:** Reducers are pure functions that update the state of the application in response to actions. Reducers take a previous state and an action as the input and return a modified version of the state. Since the state is immutable, a reducer always returns a new state, which is an updated version of the previous state.





1) Create action folder in that create showMessageReducer.js



```
JS index.js JS showMessageReducer.js JS showMessageAction.js JS App.js
1 export const SHOW_MESSAGE = "SHOW_MESSAGE";
2
3 export const GetMessage = () => {
4   console.log("DISPATCHING ACTION");
5   return {
6     type: SHOW_MESSAGE,
7     payload: { message: "Hello from Prepbytes" }
8   };
9 };
10
```

And again create a reducer folder in that create index.js and showMessageReducer.js

```
JS index.js x JS showMessageReducer.js JS showMessageAction.js JS App.js
1 import { combineReducers } from "redux";
2 import showMessageReducer from "../showMessageReducer";
3
4 const combinedReducers = combineReducers({ data: showMessageReducer });
5
6 export default combinedReducers;
7
```

JS index.js	JS showMessageReducer.js x	JS showMessageAction.js	JS App.js
1	import { SHOW_MESSAGE } from "../action/showMessageAction";		
2			
3	const showMessageReducer = (		
4	state = { message: "No message" },		
5	{ type, payload }		
6	) => {		
7	switch (type) {		
8	case SHOW_MESSAGE:		
9	console.log("STATE UPDATION");		
10	return payload;		
11			
12	default:		
13	return state;		
14	}		
15	};		
16			
17	export default showMessageReducer;		
18			

In App.js

```

import React, { Component } from "react";
import { GetMessage } from "../action/showMessageAction";
import { connect } from "react-redux";
class App extends Component {
  showMessage = () => {
    console.log("CALLING ACTION");
    this.props.getMessage();
  };

  render() {
    return (
      <div className="App">
        <header className="App-header">
          <h3>Flow of data in a React-Redux app</h3>
          <button onClick={this.showMessage}>
            Click me to trigger an action!{" "}
          </button>
          <h5>
            The updated state will be displayed here: {this.props.message}
          </h5>{" "}
        </header>
      </div>
    );
  }
}

```

```

    </h5>
    The updated state will be displayed here: {this.props.message}
  </h5>{" "}
</header>
</div>
);
}
}

const mapActionsToProps = {
  getMessage: GetMessage
};

const mapStateToProps = (state) => {
  return { message: state.data.message };
};

export default connect(mapStateToProps, mapActionsToProps)(App);

```

\*\*\*\*\*  
\*

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

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

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

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

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

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

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

```
const mapDispatchToProps = ({
  onTodoClick
})
```

\*\*\*\*\*  
\*\*\*\*

### 3.Explain context API

The React Context API is a way for a React app to effectively produce global variables that can be passed around. This is the alternative to "prop drilling" or moving props from grandparent to child to parent, and so on.

Context API is a (kind of) new feature added in version 16.3 of React that allows one to share state across the entire app (or part of it) lightly and with ease.

1. Create a folder under your app root named contexts (not required. just a convention)
2. Create a file named <your context name>Context.js, e.g. userContext.js
3. import and create context like so:

```
import React, { createContext } from "react";
const UserContext = createContext();
```

4. Create a component that will wrap the provider named Provider e.g. UserProvider  
Example using React Hooks:

```
const UserProvider = ({ children }) => {
  const [name, setName] = useState("John Doe");
  const [age, setAge] = useState(1);
  const happyBirthday = () => setAge(age + 1);
  return (
    <UserContext.Provider value={{ name, age, happyBirthday }}>
      {children}
    </UserContext.Provider>
  );
};
```

## 5.Finally export them

```
export { UserProvider, withUser };
```

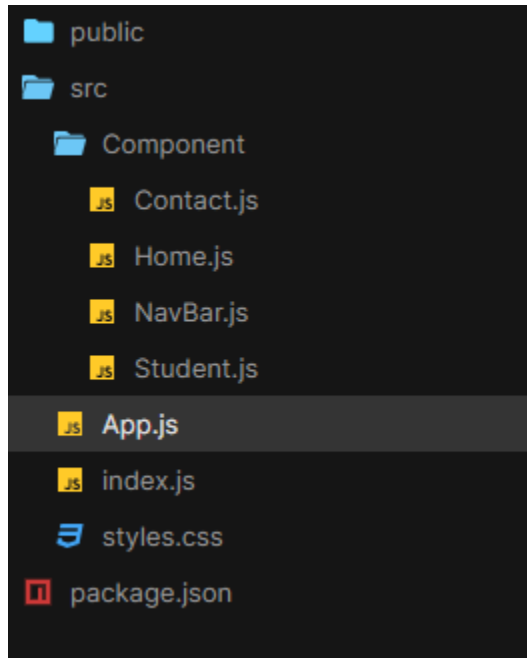
## 6.And use them however you like For example:

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

```
export default withUser(LoginForm);
```

```
*****
*****
```

#### 4. Implement the Navigation bar using a browser router ?



In App.js folder



```
JS App.js x JS Contact.js JS NavBar.js JS Student.js JS Home.js
1 import React from "react";
2 import { BrowserRouter, Routes, Route } from "react-router-dom";
3 import Contact from "../Component/Contact";
4 import Home from "../Component/Home";
5 import NavBar from "../Component/NavBar";
6 import Student from "../Component/Student";
7
8 function App() {
9   return (
10     <>
11       <div className="App">
12         <BrowserRouter>
13           <Navbar />
14           <Routes>
15             <Route path="/home" element={<Home />} />
16             <Route path="/student" element={<Student />} />
17             <Route path="/contact" element={<Contact />} />
18           </Routes>
19         </BrowserRouter>
20       </div>

```

## In Home.js

```
JS App.js JS Contact.js JS NavBar.js JS Student.js JS Home.js x
1 import React from "react";
2
3 const Home = () => {
4   return <div className="HomePage">Home</div>;
5 };
6 export default Home;
7
```

## In Student.js

```
JS App.js x JS Contact.js JS NavBar.js JS Student.js x JS Home.js
1 import React from "react";
2
3 const Student = () => {
4   return <div className="HomePage">Student</div>;
5 };
6 export default Student;
7
```

## In Contact.js

```
JS App.js JS Contact.js x JS NavBar.js JS Student.js JS Home.js
1 import React from "react";
2
3 const Contact = () => {
4   return <div className="contactPage">Contact Us</div>;
5 };
6 export default Contact;
7
```

## In Navbar.js

```

import React from "react";
import { Link } from "react-router-dom";

const NavBar = () => {
  return (
    <div className="container">
      <div class="navbar">
        <ul>
          <li>
            <Link to="/home" className="homeLink">
              Home
            </Link>
          </li>
          <li>
            <Link to="/student" className="studentLink">
              Student
            </Link>
          </li>
          <li>
            <Link to="/contact" className="contactLink">

```

```

              Contact Us
            </Link>
          </li>
        </ul>
      </div>
    </div>
  );
};
export default NavBar;

```

Output:

*Home*

*Student*

*Contact Us*

