1. **DOM**DOM = Document Object Model.   
   Document = HTML document.   
   Object = Everything inside document i.e. all HTML Tags & HTML Elements.  
   Model = Layout or Structure.  
   So DOM is the layout of the Objects or HTML tags in a document or HTML document.
2. **Virtual DOM**Exact Replica of DOM, upon which React performs all the operations.  
   When changes occur in a component's state or props, React creates a new virtual DOM, which is then compared to the previous virtual DOM to identify the changes that need to be made in the real DOM. This process is called reconciliation, and it allows React to reduce the number of DOM manipulations, and improve the performance of the application
3. **JSX**Acronym for JavaScript XML. It is HTML like code written in JavaScript.  
   Browsers can't read JSX directly. Browser understands HTML, CSS and JS. JSX is converted into JavaScript using Babel.
4. **Higher Order Component**It is a function that takes a component as an argument and returns a new component. It allows developers to reuse component logic across multiple components.  
   Use Case: - HOCs can be used to encapsulate common logic such as fetching data, managing state, or handling events, and then apply that logic to multiple components. This can help to reduce code duplication and make components more modular and reusable.

Example given on: https://blog.logrocket.com/understanding-react-higher-order-components/

1. **State**State is an Object that is used to contain data or information about the component. Whenever state changes the component re-renders.
2. **Props**  
   Props allow us to pass data from one component to another component.
3. **Children Props**Children props allow us to pass components as data to other components.  
   It is used to display whatever you include between the opening and closing tags when invoking a component. For E.g.  
   *const Picture = (props) => {  
    return (  
    <div>  
    <img src={props.src}/>  
    {props.children}  
    </div>  
    )  
   }*

*return (  
 <div className='container'>  
 <Picture key={picture.id} src={picture.src}>  
 //what is placed here is passed as props.children   
 </Picture>  
 </div>  
 )*

Instead of invoking the component with a self-closing tag <Picture /> if you invoke it will full opening and closing tags <Picture> </Picture> you can then place more code between it.

This de-couples the <Picture> component from its content and makes it more reusable.

1. **React Fragments**Fragments allow us to group a list of children without adding extra nodes to the DOM.  
   We can use <React.Fragment></React.Fragment> or its shorter syntax <></>
2. **Lifecycle method of React**  
   React contains 3 stages

* Mounting
* Updating
* Un-Mounting

Mounting contains 4 lifecycle methods: (Hint: Constructor get Re Co)

* constructor
* getDerivedStateFromProps()
* render()
* componentDidMount()

Updating contains 5 lifecycle methods: (get SCREN get COD)

* getDerivedStateFromProps()
* shouldComponentUpdate()
* render()
* getSnapshotBeforeUpdate()
* componentDidUpdate()

Un-mounting contains 1 lifecycle method

* componentWillUnmount()

1. **Proptype**It allows us to define the expected types of props that are passed to a component. PropTypes validate the props at runtime and help catch bugs and prevent unexpected behavior.

To use PropTypes, you need to import it from the "prop-types" package and define the expected types for each prop in the component. E.g.

*import PropTypes from 'prop-types';  
function Greeting(props) {  
 return <h1>Hello, {props.name}!</h1>;  
}  
Greeting.propTypes = {  
 name: PropTypes.string.isRequired  
};*

In this example, we define a prop type for the "name" prop and specify that it is expected to be a string. We also use the "isRequired" validator to ensure that the prop is passed to the component.  
PropTypes can also be used to validate objects, arrays, and other complex data structures. They support a wide range of validators, including "isRequired", "arrayOf", "objectOf", "shape", and more.

1. **Redux**Redux is a javascript library that is used for state management, thus managing the state of application.

At its core, Redux follows a unidirectional data flow (flux pattern). It is 3 components:

* **Store** holds the current state of our application.
* **Actions** are plain JavaScript objects that describe the changes to be made to the application state.
* **Reducers** are functions that specify how the application's state changes in response to dispatched actions

For installation:- *npm install redux react-redux*

*import { createStore } from 'redux';  
const initialState = {count: 0};*

*function reducer(state = initialState, action) {  
 switch (action.type) {  
 case 'INCREMENT': return {count: state.count + 1};  
 case 'DECREMENT': return {count: state.count - 1};  
 default: return state;  
 }  
}  
const store = createStore(reducer);  
export default store;*

1. **Refs**Refs allows us to get a reference to a DOM element. Refs provide a way to access and manipulate the underlying DOM nodes or child components outside of the normal React data flow.  
   We can use the useRef() hook to create refs in functional components.
2. **Reconciliation**When a component's props or state change, React compares the new values with the previous values and determines which parts of the UI need to be updated. This process is called Reconcilation.  
   React uses a diffing algorithm to compare the previous and current versions of the UI and generate a minimal set of changes that need to be applied. This allows React to avoid unnecessary updates and improve performance.  
   During the reconciliation process, React creates a new tree of React elements and compares it with the previous tree. React then determines the differences between the two trees and updates the affected parts of the UI.  
   Diffing algorithm follows a heuristic approach with complexity of O(n).  
   It is based on 2 assumptions:-

* If an element's type has changed, React assumes that the entire subtree has changed and replaces it with a new subtree.
* The developer can hint at which child elements may be stable across different renders with a "key" prop.

1. **Hooks**  
   Hooks are the functions which "hook into" or connect to React state and lifecycle features for function components.
2. **Key prop**  
   Keys help React identify which items have changed, are added, or are removed.
3. **useState**   
   useState is a Hook that allow us to add React state to function components.
4. **useEffect**It allows us to perform side effects in response to changes in props, state, or other variables.
5. **useMemo**It is used to memoize a value, which means that the value is only recomputed when its dependencies change.  
   The useMemo() hook takes two arguments: a function that computes the value, and an array of dependencies that the value depends on. The function is only re-run when one of the dependencies changes, and the memoized value is returned from the hook.

*import React, { useMemo } from 'react';  
function MyComponent(props) {  
 const { a, b } = props;  
 const result = useMemo(() => {  
 console.log('Computing result...');  
 return a + b;  
 }, [a, b]);  
return (  
 <div>  
 <p>Result: {result}</p>  
 </div>  
 );  
}*In this example, the useMemo() hook is used to compute the sum of ‘a’ and ‘b’, and the resulting value is stored in the result variable. The function passed to useMemo() is only re-run when either ‘a’ or ‘b’ changes, and the memoized result value is returned from the hook.

1. **useRefs**It is a hook used to create refs.
2. **20. useCallback**It is a hook that allows you to memoize a function, which means that the function is only recreated when its dependencies change.  
   The useCallback() hook takes two arguments: a function to memoize, and an array of dependencies that the function depends on. The memoized function is returned from the hook, and can be passed as a prop or used in other parts of your component.  
   *import React, {useState,useCallback } from 'react';*  
    *function MyComponent() {  
    const [count,setCount] = useState(0);  
    const incrementBy1 = useCallBack(()=>{  
    setCount(count+1);  
    },[count]);  
   return (  
    <div>  
    <button onClick={incrementBy1}>Click me</button>  
    </div>  
    );  
    }*

*The button, "Click Me," uses the IncrementBy1 function as the click handler. useCallback ensures that the IncrementBy1 function retains the same reference across renders as long as its dependencies (in this case, [count]) remain unchanged. This prevents unnecessary re-creation of the function when the component re-renders.*

1. **Creating Custom Hook***import { useState, useEffect } from 'react';  
   function useFetch(url) {  
    const [data, setData] = useState(null);  
    const [error, setError] = useState(null);  
    const [loading, setLoading] = useState(true);  
    useEffect(() => {  
    const fetchData = async () => {  
    try {  
    const response = await fetch(url);  
    const json = await response.json();  
    setData(json);  
    setLoading(false);  
    } catch (error) {  
    setError(error);  
    setLoading(false);  
    }  
    };  
    fetchData();  
    }, [url]);   
    return { data, error, loading };*

*}  
export default useFetch;*

In this example, the useFetch() hook is created to fetch data from an API endpoint and return the data, error, and loading status.  
The useFetch() hook is then exported as a module, and can be used in other components.

*import useFetch from './useFetch';  
function MyComponent() {  
 const { data, error, loading } = useFetch('https://api.example.com/data');  
 if (loading) {return <div>Loading...</div>;}  
 if (error) {return <div>Error: {error.message}</div>; }  
return (  
 <div>  
 {data && JSON.stringify(data)</pre>}  
 </div>  
 );*

*}*

1. **When does React Component re-renders?**In React, a component re-renders when its state or props change.
2. **Context API**In React, the Context API is a feature that allows us to share data between components without passing the data down through props. Context provides a way to avoid "prop drilling", where props are passed through many levels of components to reach a deeply nested component.
3. **Context API v/s Redux**Here are some of the key differences between the Context API and Redux:

* Complexity: The Context API is simpler and easier to use than Redux, making it a good choice for small to medium-sized applications. Redux, on the other hand, has a steeper learning curve and requires more setup, but can provide more advanced features for larger applications.
* Centralized Store: Redux includes a centralized store that holds all of the application's state, while the Context API allows you to create multiple context objects to store state in different parts of the application.
* Middleware: Redux includes middleware, which allows you to intercept and modify actions and add additional functionality to the store. The Context API does not include middleware.
* Debugging: Redux includes time travel debugging, which allows you to step back and forth through the application's state changes, making it easier to debug and understand complex applications.

Overall, the Context API can be a good choice for small to medium-sized applications with a limited amount of shared state, while Redux is better suited for larger and more complex applications with a significant amount of shared state and advanced state management needs. However, the choice between the two ultimately depends on the specific requirements and complexity of the application.

1. **Pure Components**Pure components are a type of component that only re-render when their props or state change, which can help reduce unnecessary re-renders and improve the overall efficiency of a React application. Pure components can be created using the React.PureComponent class or by using the React.memo higher-order component.
2. **Error Boundries**Error Boundries are react components that   
    a. catches javascript errors anywhere in their child component tree,  
    b. log those errors.  
    c. display a fallback UI instead of component tree that crashed

A class component becomes an error boundary if it defines either (or both) of the lifecycle methods static getDerivedStateFromError() or componentDidCatch().  
getDerivedStateFromError() to render a fallback UI after an error has been thrown. componentDidCatch() to log error information.

1. **Portals**Portals provide a way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.  
   We can create portal using ReactDOM.createPortal  
   e.g. ReactDOM.createPortal(<child-component>,target-dom-node)  
   ReactDOM.createPortal takes two arguments: the child component to render, and the target DOM node where the child component should be mounted.
2. **Fetching data from Service**Get Request*fetch("url").then(response=>{console.log(response)}).catch(error => {console.log(error)})*

Post Request *fetch('https://example.com/profile', {  
 method: 'POST', // or 'PUT'  
 headers: { 'Content-Type': 'application/json' },  
 body: JSON.stringify(data),  
}).then(response => { console.log(response)}).catch(error => { console.log(error)})*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Get request  
*axios.get(URL).then((response) => {console.log(response.data)});* Post Request  
*axios.post(URL, {  
 title: "Hello World!",  
 body: "This is a new post."  
}).then((response) => {console.log(response.data)});* Axios is better than fetch because

1. Axios has better error handling. It throws 400 and 500 range errors for us. Unlike the Fetch API, where we have to check the status code and throw the error yourself.
2. With axios we don't need to set headers and converting request body to JSON string. Axios do all these things for us which lacks in Fetch.
3. **Controlled v/s Uncontrolled Component  
   Controlled Component** is one that takes its current value through props and notifies changes through callbacks like "onChange"  
   In a controlled component, the value of the input element is controlled by React.  
   for example:- <input type="text" value={value} onChange={handleChange} />

**Uncontrolled Component** query the DOM using a "ref" to find its current value when we need it.  
In an uncontrolled component, the value of the input element is handled by the DOM itself.  
 for example:- <input type="text" defaultValue="foo" ref={inputRef} />  
// Use `inputRef.current.value` to read the current value of <input>

1. **Strict Mode**  
   Strict Mode is used to highlight potential problems in an application. It performs additional checks on the application.  
   StrictMode can helps with:

* Identifying components with unsafe lifecycles
* Warning about legacy string ref API usage
* Warning about deprecated findDOMNode usage
* Detecting unexpected side effects
* Detecting legacy context API
* Ensuring reusable state

1. **Lazy Loading.**
2. Lazy Loading is also called Code Splitting.
3. React has a concept of Component Based Architecture which means that instead of building our application as a whole we can split our application into multiple reuseable components.
4. Now when we do production build at that time webpack(under the hood) will bundle all of the component and it will generate bundle.js file.
5. On initial request, If our application is huge then the JS bundle will take a long time to download & execute the script file which in turn lead to performance loss.
6. To avoid the performance loss we use code splitting. React Lazy & Suspense are preferred way to do code splitting in react applications.
7. React.lazy taskes a function that must call a dynamic import().
8. The lazy component should then be rendered inside the suspense component, which allow us to show fallback content(such as loading indicator) while we wait for lazy component to load.

For Example:-

* Let's assume we have a "Profile" component and we have applied code splitting.
* Whenever we visits "Profile" page at that time React will dynamically load that component.
* It simply means that code for "Profile" component is not initially included in initial "bundle.js" file that gets sent to the client.
* Instead, a seperate chunk file is created for the Profile component which only gets downloaded when user visits that component.

Syntax

Lazy Loading using <Suspense>

1. Import the component that will be loaded lazily. e.g. const UserProfile = React.lazy(() => import('./UserProfile.js'));

2. Wrap the imported component with <Suspense>.   
 <Suspense fallback = {<p>Loading....</p>}> <UserProfile/> </Suspense>.

3. fallback prop accepts a component that will be visible until the actual component loads.

1. **Authentication in React js.**

One of the promising and best way is to use JWT. We will store the JWT token & store it in local storage. This token can be appended in header to every or some axios request in Header.

Case 1: If we want to append the token to every request.

*axios.interceptors.request.use(request=>{  
 request.headers.authorization = localStorage.getItem("jwt\_token");  
 });*

Case 2 : If we want to append token to only some requests

*const authAxios = axios.create({  
 baseUrl: apiUrl,  
 headers: { Authorization: localStorage.getItem("jwt\_token") }  
 })*

1. **Storing & fetching data in local storage & session storage** localStorage.setItem("jwt\_token", token);  
    localStorage.getItem("jwt\_token");  
    localStorage.clear();

sessionStorage.setItem("jwt\_token", token);  
 sessionStorage.getItem("jwt\_token");  
 sessionStorage.clear();

1. **What are Interceptors**Interceptors are the default configurations that are added automatically to every api request or response that a user receives.
2. **Routing in React js**

React-Router is collection of Router Components, Route Matching components and Navigation Components.

* Router Components: <BrowserRouter>
* Route Matching Components : <Route> and <Switch>
* Navigation Components : <Link>

*import { BrowserRouter, Route, Switch, Link } from 'react-router-dom'.*

* <BrowserRouter>, is usually given an alias of ‘Router’ and this is the parent component that is used to store all of your <Route> components.
* <Route> tells our app which other components to display based on the route.
* <Switch> ensure that only one route is rendered at a time. Order of route matters in Switch.
* <Link> components are how you create links to those different routes.

Example  
import React from 'react';  
import ReactDOM from 'react-dom';  
import { BrowserRouter as Router, Route, Switch, Link } from 'react-router-dom';  
import Home from './components/Home';  
import About from './components/About';

const App = () => {  
 return (  
 <Router>  
 <nav>  
 <ul>  
 <li> <Link to="/">Home</Link> </li>  
 <li> <Link to="/about">About</Link> </li>  
 </ul>  
 </nav>  
 <Switch>  
 <Route exact path="/" component={Home} />  
 <Route path="/about" component={About} />  
 </Switch>  
 </Router>  
); }

1. **Form Validation**

We are implementing form Validation using controlled components.

We also use formik library.

1. Where should we store our JWT token in React JS application?

* **In Memory**: We can store the JWT token in memory as a JavaScript variable or in a state variable within React components. However, this is not recommended when we need to persist the token across sessions, as the token will be lost if the user refreshes the page or closes the browser.
* **In Browser Cookies**: We can store the JWT token in an HTTP cookie. Cookies provide a secure and persistent way to store tokens, and they are automatically sent with each HTTP request to the server and can have attributes like ‘secure’(for HTTP only cookie) for added security.
* **In Browser Local Storage or Session Storage**: You can store the JWT token in the browser's local storage or session storage. However, this approach is not recommended for sensitive tokens due to potential security risks, such as cross-site scripting (XSS) attacks. Use this option only if you have specific reasons to do so.
* **In Redux Store or Context (State Management)**: If you are using a state management library like Redux. This allows you to access the token from any component within your application. Cons: Requires additional setup for state management, and tokens may not persist across page refreshes.

1. How to get expiry time of JWT?

We can use claims in JWT.

In a JWT (JSON Web Token), claims are statements about an entity (typically, the user) and additional data. Claims provide information about the entity, such as their identity, roles, and metadata. JWT tokens consist of three parts: a header, a payload, and a signature. Claims are included in the payload part of the token. There are three types of claims: registered claims, public claims, and private claims.

**Registered Claims**: These are a set of predefined claims that are recommended to be consistently used across different applications. They are not mandatory but are commonly recognized and have a standard meaning. Some of the common registered claims include:

* iss (Issuer): The entity that issued the JWT.
* sub (Subject): The subject of the JWT (typically, the user).
* aud (Audience): The intended audience of the JWT.
* exp (Expiration Time): The expiration timestamp of the JWT.
* nbf (Not Before): The timestamp before which the JWT is not valid.
* iat (Issued At): The timestamp indicating when the JWT was issued.
* jti (JWT ID): A unique identifier for the JWT.

**Public Claims**: These are custom claims defined by the users or organizations that create the JWT. Public claims can be defined to carry information relevant to the application's specific use case. For example, you might include claims like username, email, role, or any other user-related data.

**Private Claims**: Private claims are custom claims defined by the parties that exchange the JWT and are intended to be recognized and processed by those parties only. Private claims are not registered and should not be used for public interoperability. They are typically used for application-specific information or data shared between parties that understand the semantics of the claim.

1. How to store JWT token in cookies

import Cookies from 'js-cookie';

// Example: Receive a JWT token during user authentication

const jwtToken = 'your-jwt-token'; // Replace with your actual JWT token

// Store the JWT token in a secure HttpOnly cookie

Cookies.set('jwtToken', jwtToken, { secure: true, sameSite: 'strict', expires: 7 });

In this example, we import Cookies from ‘js-cookie’ library and use it to set a cookie named 'jwtToken' with the JWT token as the value.

We set the secure option to true to ensure that the cookie is only sent over HTTPS connections, enhancing security.

The sameSite option is set to 'strict' to control when the cookie is sent in cross-origin requests.

The expires option specifies the cookie's expiration time in days. Adjust this value according to your application's requirements.

**Retrieve the JWT token from the cookie**

const jwtToken = Cookies.get('jwtToken');

We can implement Single Sign on application with cookies.

1. What are the optimization techniques in react js.

* Use of useMemo and use CallBack Hooks for memorizing and thus preventing unnecessary re-renders.
* Use Lazy Loading. Using dynamic imports.
* Using React.memo to create Pure components.

1. How to error handling in React JS application

* Using try catch block.
* Using Error Boundries.
* In case of Http requests use .then().catch()
* Global Error Handling: use JavaScript's global window.onerror event or window.addEventListener('error') to capture unhandled errors on the entire page.

1. Equivalence of lifecycle methods in class base components to react hooks

* **componentDidMount Equivalent**: In class-based components, componentDidMount is used to perform actions after the component has been inserted into the DOM. The equivalent in functional components is the useEffect hook with an empty dependency array. This hook runs once, after the initial render:

useEffect(() => {

// Your code here

}, []);

* **componentDidUpdate Equivalent**: componentDidUpdate is called after a component's state or props change. The equivalent in functional components is the useEffect hook with a dependency array that specifies the variables to watch for changes:

useEffect(() => {

// Your code here

}, [dependency1, dependency2]);

* **componentWillUnmount Equivalent :**  componentWillUnmount is used to clean up resources or cancel ongoing processes when a component is unmounted. The equivalent in functional components is the useEffect hook's cleanup function:

useEffect(() => {

// Your code here

return () => {

// Cleanup code here

};

}, []);

* **shouldComponentUpdate Equivalent**: shouldComponentUpdate is used to optimize rendering by preventing unnecessary updates. Returning false from this method prevents the component from updating. The equivalent in functional components is the React.memo higher-order component. You can wrap your component with React.memo to achieve similar optimization:  
  const MemoizedComponent = React.memo(MyComponent);