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.

Example: React.memo is HOC which is used to make Pure 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 library that is used for 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, so that we can access and manipulate the underlying DOM elements 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.  
   It takes 2 arguments a callback function and a dependency array.  
   If there is no dependency array then useEffect’s callback function will be called on every render.  
   If there is empty dependency array then useEffect’s callback will be called only once.
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. **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.

Example:

**Creating the context in Context.js file**

Import {createContext} from “react”;

Const UserContext = createContext({

loggedInUser: “DefaultUser”

});

export default UserContext;

Using the above context using useContext hook.

Const {loggedInUser}= useContext(UserContext);

**To change the value of UserContext**

<UserContext.Provider value = {{loggedInUser: “New User”}}>

//place components where we have to use UserContext

</UserContext.Provider>

1. **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, making it a good choice for small to medium-sized applications. Redux, on the other hand 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 us to asynchronously *monitor 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.
* Redux is easily scalable as compared to React.

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**It renders 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=>response.json()).then(result=>console.log(result)).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=>response.json()).then(result => { console.log(result)}).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, Chunking, Dynamic Bundling, etc.
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>  
); }

React Router 6

Use react-router-dom library.

Import {createBrowserRouter, RouterProvider} from ‘react-router-dom’;  
const appRouter = createBrowserRouter([  
 {  
 path: “/”,  
 element: <Home/>,  
 errorElement: <Error/>  
 },  
 {  
 path: “/about”,  
 element: <About/>  
 },  
 {  
 path: “/contact”,  
 element: <Contact/>  
 }  
]);

We also have useRouteError which gives more information about error. It has to be used in component corresponding to errorElement.

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. Claims are part of payload.

In a JWT, you typically need to decode the token to access the claims contained within it.

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.
* Use key prop wherever necessary.
* Follow single responsibility principle.

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

};

}, []);

Explanation:

The function that you return inside the useEffect is often referred to as the "cleanup function" or "cleanup handler." It's a critical part of managing side effects in a functional component. This function is called when the component is unmounted or when the dependencies specified in the dependency array change in a way that would trigger a cleanup.

Here's a detailed explanation of how the cleanup function works:

**When the Component Mounts:**

When the component mounts, the code block inside the useEffect is executed. This code block usually contains setup logic or side effects you want to run when the component is mounted. It might include things like setting up event listeners, starting timers, making network requests, or performing any other side effect you need in your component.

**When the Component Unmounts:**

If the component unmounts (i.e., it is removed from the DOM), the cleanup function is called automatically by React before the component is removed.

The purpose of the cleanup function is to handle any necessary cleanup or teardown for the side effects that were set up when the component mounted. For example, if you added event listeners or started timers in the setup code, the cleanup function is where you would remove those event listeners or clear those timers.

**When Dependency Values Change:**

If you specify a variable as a dependency, and that variable changes, the cleanup function will be invoked before the effect with the updated dependency runs. This is helpful for situations where you need to clean up previous side effects before applying new ones based on changed dependencies.

* **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);

New Features in React 18.

* Automatic Batching : State modifications done using event handler are grouped together.

Eg. Function handleClick() {  
 setIsFetching(false);  
 setError(null);  
 setFormStatus(“Success”)  
}

* Transitions: Used to distinguish between urgent and non urgent updates.
* Suspense on the server: Provide calm loading state while user is dealing with network conflict.
* Concurrent Rendering: React can interrupt, pause, restart or quit a render in React 18 with concurrent rendering.

New Hooks Introduced in React 18.

* useId
* useTransition
* useDefferedValue
* useSyncExternalStore
* useInsertionEffect

Client Side Changes in React.

Earlier Implementation  
 import React from ‘react’;  
 import ReactDOM from ‘react-dom’;  
 import App from ‘./App’;  
 ReactDom.render(  
 <React.StrictMode><App/><React.StrictMode>  
 document.getElementById(“root”);  
 );

Present Implementation  
 import React from 'react';  
 import ReactDOM from 'react-dom/client';  
 import App from ‘./App’;  
 const rootElement = document.getElementById(“root”);  
 const root = ReactDOM.createRoot(rootElement);  
 root.render( <React.StrictMode><App/><React.StrictMode>);

Hydration in react

Redux Saga.

Redux-saga library that is used to handle asynchronous actions in a Redux application.

Note: In the context of Redux Saga, generator functions are often referred to as sagas.

**Difference between Redux Thunk and Redux Saga**

Redux Thunk and Redux Saga are two popular middleware libraries used with Redux to handle asynchronous actions and side effects in Redux applications. They serve similar purposes but have different approaches and characteristics. Here are some key differences between Thunk and Saga:

**Handling Asynchronous Actions**

Redux Thunk: Redux Thunk is a middleware that allows you to write action creators that return functions instead of plain action objects. These functions can contain asynchronous logic and dispatch actions when the asynchronous operations are complete. Thunks are straightforward and easy to grasp, making them a good choice for simple asynchronous flows.  
Redux Saga: Redux Saga is a middleware library that uses generator functions to handle asynchronous actions. Sagas allow you to express complex asynchronous flows as a sequence of actions. They provide more control over the flow of your application and can handle advanced use cases such as cancellation, debouncing, and complex control flow.

**Complexity and Control**

Redux Thunk: Thunks are simple and have less built-in complexity. They are a good fit for basic asynchronous operations like making API calls. However, they may not be as suitable for more complex scenarios or situations requiring fine-grained control.  
Redux Saga: Sagas offer more control and are designed for handling complex asynchronous tasks. You can use various built-in effects, such as takeLatest, takeEvery, and take, to manage the flow of actions. Sagas are particularly well-suited for scenarios like real-time updates and WebSockets.

**Testing**

Redux Thunk: Testing thunks is relatively simple because thunks are functions that you can call directly, and you can easily mock the API calls within them for testing purposes.  
Redux Saga: Testing sagas can be more complex due to the generator function nature of sagas. You may need additional libraries or utilities to test sagas effectively, such as redux-saga-test-plan.

**What are effects in Redux Saga?**

Effects in Redux Saga are objects that describe the operations you want to perform within your generator functions (sagas). These operations can include things like making API requests, dispatching actions, delaying, and more. Effects are used to handle asynchronous and side effect-related tasks. Some common effects include put, call, take, takeEvery, and takeLatest, etc.

**What are watchers in Redux Saga?**

Watchers in Redux Saga are sagas(generator functions) responsible for watching and responding to specific actions or events. They act as the entry points for our sagas and are typically used to initiate the execution of other sagas. Watchers use effects like take, takeEvery, or takeLatest to wait for specific actions to be dispatched.

Watcher sagas use effects like take, takeEvery, takeLatest, or similar effects to listen for specific action types. They await these actions to be dispatched.

Example of Watcher in next line.

*import { takeLatest } from 'redux-saga/effects';  
function\* watchFetchData() { yield takeLatest('FETCH\_DATA', fetchDataSaga);}*

**What are Workers in Redux-Saga?**

Workers typically refer to the individual generator functions that perform specific tasks and handle the core business logic or side effects within your Redux Saga application.

Worker sagas use effects like put, call, select, etc., to perform actions such as dispatching new actions, making API requests, and retrieving data from the Redux store.

**Enlist some of the built in effects in saga?**

1. take(actionType): Listens for a specific action (actionType) to be dispatched and pauses the generator until that action is dispatched.
2. takeEvery(actionType, saga): Listens for a specific action (actionType) and starts a new instance of the saga each time that action is dispatched.
3. takeLatest(actionType, saga): Listens for a specific action (actionType) and automatically cancels any previously started instances of saga if a new action of the same type is dispatched.
4. takeLeading(actionType, saga): Listens for a specific action (actionType) and starts a new instance of saga only if no other instances are currently running.
5. takeMaybe(actionType, saga): Listens for a specific action (actionType) and starts a new instance of saga only if there are no other instances currently running.
6. put(action): Dispatches a Redux action to the store.
7. call(fn, ...args): Calls a function fn with the specified arguments and waits for the function to complete. Useful for handling asynchronous operations.
8. apply(context, fn, args): Calls a function fn with the specified context and arguments and waits for the function to complete.
9. all([...effects]): Runs multiple effects in parallel and waits for all of them to complete. Returns an array of results.
10. race({ effect1, effect2, ... }): Runs multiple effects in parallel and resolves with the result of the first effect to complete.
11. delay(milliseconds, value): Delays the saga by the specified number of milliseconds and then resolves with the optional value.
12. select(selector, ...args): Retrieves data from the Redux store using a selector function. You can also pass additional arguments to the selector function.
13. fork(saga, ...args): Forks a new task that runs the given saga. It does not block the current saga and runs the forked task in the background.
14. spawn(saga, ...args): Similar to fork, but the spawned task is considered independent, and any errors do not bubble up to the parent saga.
15. cancel(task): Cancels a forked or spawned task.
16. join(task): Pauses the saga until the specified task is complete.
17. actionChannel(actionType, buffer): Creates a channel for a specific action type, allowing you to buffer and control the flow of actions.
18. flush(channel): Empties a channel, resolving with all items in the channel at the time of flushing.
19. cancelled: A special effect that resolves if the current saga was canceled.
20. getContext(propName): Retrieves a property from the saga's context.

**What Does takeLatest do?**

Here's what takeLatest does:

* **Listening for Actions**: takeLatest is a function that listens for a specific Redux action (typically a "trigger" action) to be dispatched in your application. It does this using a generator function.
* **Cancellation of Previous Sagas**: When a new action of the same type is dispatched while the previous one is still in progress, takeLatest will automatically cancel the previous running saga and start a new one to handle the latest action. This is particularly useful for situations where you only want to handle the most recent request and ignore older requests. For example, in the context of making an API call, you might want to cancel the previous API call if the user triggers a new one before the previous one is complete.
* **Example Use Case:** A common use case for takeLatest is handling network requests. For instance, if you have an action FETCH\_DATA, and you want to ensure that only the most recent fetch request is active at any given time, you can use takeLatest to handle this.

Here's a basic example of how you might use takeLatest in a Redux-Saga:

*import { takeLatest, call, put } from 'redux-saga/effects';  
import { fetchDataSuccess, fetchDataFailure } from './actions';  
import { api } from './api';  
function\* fetchMyData(action) {  
 try {  
 const data = yield call(api.fetchData, action.payload);  
 yield put(fetchDataSuccess(data));  
 } catch (error) {  
 yield put(fetchDataFailure(error));  
 }  
 }*

*function\* mySaga() {  
 yield takeLatest('FETCH\_DATA', fetchMyData);  
}  
export default mySaga;*

In this example, takeLatest listens for the 'FETCH\_DATA' action. If a new 'FETCH\_DATA' action is dispatched while fetchMyData is already running (e.g., the user clicks a "Refresh" button multiple times quickly), the previous running fetchMyData saga will be canceled, and a new one will start with the latest action. This ensures that only the most recent data fetch request is active at any given time.

**What does call do in redux-saga?**

In Redux Saga, the ‘call’ function is used to call functions

* that return promises or
* that are asynchronous in nature within our generator functions.

*import { call } from 'redux-saga/effects';  
function\* mySaga() {  
 try {  
 const result = yield call(asyncFunction, arg1, arg2);  
 // 'result' will contain the resolved value of 'asyncFunction'  
 } catch (error) {  
 // Handle any errors that occur during the asynchronous operation  
 }  
}*

Here's how the call function works:

* We pass an asyncFunction as first argument to redux-saga’s ‘call’ function followed by any number of arguments that should be passed to that asyncFunction.
* Inside the generator function, when yield call(asyncFunction, arg1, arg2) is executed, it effectively says, "Call asyncFunction with arg1 and arg2, and wait for it to complete before proceeding."
* If asyncFunction returns a promise, call will wait for that promise to be resolved and return the resolved value. If asyncFunction throws an error, it will be caught by the surrounding try...catch block, allowing you to handle any errors gracefully.

**What does ‘put’ do in redux-saga?**

The put function is used to dispatch Redux actions from within our generator functions.

*import { put } from 'redux-saga/effects';  
function\* mySaga() {  
 yield put({ type: 'INCREMENT' });  
 // Dispatches an action with type 'INCREMENT' to the Redux store  
}*

Namaste React

React CDN Links

<script crossorigin src="https://unpkg.com/react@18/umd/react.development.js"></script>

<script crossorigin src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"></script>

* CDN is a geographically distributed group of servers that caches content close to end users.
* The crossorigin attribute, valid on the <audio>, <img>, <link>, <script>, and <video> elements, provides support for CORS, defining how the element handles cross-origin requests, thereby enabling the configuration of the CORS requests for the element's fetched data.

Why we have 2 links?

* React.development.js is core file of react which contains all the react algorithms.
* React-dom.development.js is required for performing DOM operations.

Note: These 2 CDN should be at the top level scripts.

Creating an element and injecting it into react.

* const heading = React.createElement(“h1”, {id: “heading”}, “Hello World from React”);
* const root = ReactDOM.createRoot (document.getElementById(“root”));
* root.render(heading);

Step 1 : react.createElement creates a react element which internally is JavaScript object. It is not the H1 tag yet.

Step 3: render method takes the react element and convert it into heading tag and put it up in the DOM.

Rendering Nested element in React.

Const parent = React.createElement(  
“div”, {id: “parent”}, React.createElement(  
“div”, {id: “child”}, [  
React.createElement(“h1”, {}, “I’m a h1 heading”), React.createElement(“h2”, {}, “I’ m h2 heading”)  
]));

The above react code will produce HTML structure like below:

<div id = “parent”>  
 <div id = “Child”>  
 <h1>I’m a h1 heading</h1>  
 <h2>I’m a h2 heading</h2>  
 </div>  
</div>

If we want to draw conclusions from this we can see that React.createElement can make a highly complex code, to avoid this JSX was introduced.

Note1: Whatever we write in root tag (<div id = “root”></div>) in HTML will be replaced by whatever we pass in render method.

Note2: We can apply react to small part of existing page.

Ep2: Igniting The App.

Package.json is used for managing the dependencies and used by npm. It keep track of dependencies used in our App.

Tilde(~) and caret(^) in package.json

Caret upgrades the minor version only. Tilde upgrades the major and minor versions. It is preferred to use Caret only.

Package-lock.json

It stores an exact, versioned dependency tree rather than using starred versioning like package.json itself (e.g. 1.0.\*). This means you can guarantee the dependencies for other developers or prod releases, etc. It also has a mechanism to lock the tree but generally will regenerate if package.json changes.

NPM v/s NPX

Npm is a tool that use to install packages. Npx is a tool that use to execute packages.

Parcel/ Webpack/Vite features (All are bundlers)

Creating Dev Builds

Local Server

HMR – Hot Module Replacement

File Watching Algorithm (written in C++)

Caching (Faster Builds)

Image Optimization

Minifications

Bundling

Compress

Consistent Hashing

Code Splitting

Differential Bundling

Diagnostic

Error Handling

HTTPs

Tree Shaking Algorithm (removes unused code)

BrowsersList: Give browser specific details

Tailwind CSS

Tailwind internally uses PostCss. PostCss is a tool for transforming CSS with JavaScript.

Redux has 2 libraries

1. React-Redux (Acts as Bridge between React and Redux ToolKit)
2. Redux Toolkit (RTK)