**JavaScript:** is a synchronous single-threaded language means it can only execute one line at a time in a specific order. Everything in JavaScript happens inside an Execution Context.

Var (function scoped) is accessible inside a function in which it is defined.

Let (block scoped) is only accessible inside a block in which it is defined.

Const (block scoped) is used to define constant.

**Functions (Arrow Function and Higher Order Functions):**

**Arrays & Objects (Array De-structuring, Object De-structuring, Rest Operator, Spread Operator):**

**Conditions (if else, ternary operator, optional chaining, && and ||):**

**Array Methods (Map, Filter, Reduce, Sort):**

**Even Listeners (onClick, onBlur, setTimeOut, setTimeInterval):**

**Async Events (Callback, Promises):** We are passing a callback function to another function. Callback results in callback hell (nested callbacks, so code is unmaintainable) and inversion of control (we give control of a function to another function).

A Promise is an object representing the eventual completion or failure of an async operation. We are attaching a callback function to a promise object (pending, fulfilled, or rejected state). Promise API will return us a promise which is nothing but an empty object and after completion of its task, this empty object will be filled with data automatically after whatever async time it takes. We will attach a callback function which is .then(). We can also attach a failure callback function using .catch().

JS engine will not wait for promise to be resolved and continue to executes next line.

**Async Await:** async function always return a promise. Even if return value is not a promise (i.e., some string, etc.), it will automatically wrap it in a promise and then returns a promise.

Async await combination is used to handle promises. We use keyword await in front of a promise that has to be resolved. Await keyword can only be used inside an async function.

JS engine waits for promise to be resolved on await keyword and then executes next line.

**Try Catch:**

**Encapsulation 1/4:** We group related variables and functions that operate on them into objects. It reduces complexity and increase reusability.

**Abstraction 2/4:** We hide the implementation details and complexity, and show only essentials.

**Inheritance 3/4:** To eliminate redundant code**.**

**Polymorphism 4/4:** That allows you to eliminate/refactor long if-else or switch-case statements.

**TypeScript:** is a programming language build on top of JavaScript and to address shortcomings of JavaScript. It is statically-typed (int variable cannot be made string).

**React:** is a JavaScript library for developing and designing complex mobile UI and web applications which works on Virtual DOM, View Oriented, Unidirectional Data Flow and Component Based. It uses Component based approach which ensures to help you build components that possesses high reusability.

**JSX**: JavaScript XML which is to imbed JavaScript in html. This ensures that the resulting will have high readability, thereby relatively increasing the performance of the application.

**Router**: helps to have multiple pages in a Single Page Application. It keeps our UI in sync with URL.

Role Based Access Controller RBAC:

Protected Routes:

Dynamic Routing:

**Props**: is used to pass data between different components.

**State**: is a JavaScript object used to represent information in or about a component.

**Redux**: is used to store the state of the application in a single entity. It consists of four components which are Action (an object that describes the call), Reducer (the state change storage unit), Store (the state and object tree storage) and View (displays data provided by store).

**Components**: A JavaScript class or function that returns back some html. Components can be nested as deep as you want.

**Lifecycle Methods of Component**: Mounting (when component added to DOM), Updating (when modifying something in the component), Unmounting (when component is removed from DOM)

**Hooks**: It helps to add/use state and other React features without using class-based components. Before hooks, functional components cannot hold state. Hooks are only called inside functional components, at the top level of a component and cannot be conditional.

**useState**: allows us to track state in a function component. It returns an array of two elements, like current state and function that updates the state, like name, setName

**useEffect**: allows you to perform side effects in your components. It helps to use the lifecycle methods in the function component which takes a call back as first argument and second argument is array of properties that you want to check if changed (function and dependency).

**useContext**: is a way to manage state globally. For example, we have many nested components. The component at the top and bottom of the stack need access to the state. To do this without Context, we will need to pass the state as "props" through each nested component. This is called "prop drilling".

**useReducer**: is similar to the useState Hook. It allows for custom state logic. If you find yourself keeping track of multiple pieces of state that rely on complex logic, useReducer may be useful.

**useMemo**: returns a memoized value. Think of memoization as caching a value so that it does not need to be recalculated. The useMemo Hook only runs when one of its dependencies update. This can improve performance.

**useCallback**: returns a memoized callback function. Think of memoization as caching a value so that it does not need to be recalculated. This allows us to isolate resource intensive functions so that they will not automatically run on every render. The useCallback Hook only runs when one of its dependencies update. This can improve performance.

**useRef**: allows you to persist values between renders. It can be used to store a mutable value that does not cause a re-render when updated. It can be used to access a DOM element directly.

**Custom Hooks:** Make code Clean, Maintainable, Readable and Reusable. (Like use local storage hook).

**Higher Order Components (**Used in Machine Coding Round**):** A function which takes another function as an argument or returns a function from it.

**State** **Management**: Global state management is done by Context API or using third party library like Redux, Redux Toolkit (RTK), Zustand. With these, we are able to create some global state management.

Props Drilling:

**Lazy Loading:** (Code Splitting, Chunking, Suspense?)

**Virtual** **DOM**: is a simple JavaScript object and React creates a Virtual DOM which is a virtual representation of Real DOM. When we are updating our components, we are actually updating virtual DOM, and not the Real DOM. Using this method, React can find the most efficient way to update the real DOM by updating only areas where changes have been made without having to update the entire DOM.

Reconciliation Algorithm:

React Fiber:

Differ Algorithm:

Render Working:

**Server-Side Rendering (SSR) and Client-Side Rendering (CSR):**

**Async Tasks:**

API Calls (Fetch, or Axios):

Events:

Promises:

**Styling (Tailwind, StyleX, Bootstrap, Material UI, Ant UI):**

**Testing (1/4):** I am using higher order components for this purpose and it is very much testable also and I can write easy test cases for this. (Reusability, Modularity, Testability, Readable)

**Performance (ShimmerUI) 2/4:**

Lazy Loading:

Asset Optimization:

JS Code Optimization:

CSS Code Optimization:

Server Optimization:

Write Optimized Code:

Bundlers:

Rendering:

**Accessibility 3/4:**

**Security 4/4:**

**Key** **Prop**: should be unique and helps React identify which items have been changed, added or removed, so React knows which part of DOM has to update.

**Event** **Listeners**: We camelCase event functions and we call the function that we want directly inline between two curly braces. onClick={function1}, onSubmit={function2}

**Forms**: As we keep all information in state inside a component. In React, we typically add EvenListeners to each field and update our component state whenever any of these inputs changed. So, methods like onChange and onSubmit will directly update our state and would be controlled by our own functions, instead of letting the form handle all of this on its own.

**Conditional** **Rendering**: We will need to render some content conditionally depending on other values inside our application. Inline if else with conditional operator.

**Common** **Commands**:

npx create-react-app MyApp (creates our react application)

npm start (starts up our development server)

npm run build (creates an optimized build of our app for deployment)

**Node:** Node is a runtime environment for executing JavaScript code on a Server.

**Python:**

**AWS:**

EC2: Infrastructure as a Service

S3:

**CICD:**