**React**

**What is react?**

React is an open-source JavaScript library used for building user interfaces (UIs) for web and mobile applications. It was developed and is maintained by Facebook, and it is often used for creating interactive and dynamic front-end applications.

**Features:**

1. **Virtual DOM**: React's virtual DOM efficiently updates the actual DOM, reducing unnecessary re-rendering and improving performance. This feature is particularly beneficial for applications with complex user interfaces.
2. **Component-Based Architecture:** React promotes a modular and reusable component-based architecture, making it easier to maintain and scale applications. Components can be composed together to build complex UIs.
3. **Declarative Syntax:** React uses a declarative syntax, allowing developers to describe how the UI should look based on the application's state. This makes code more predictable and easier to understand.
4. **One-Way Data Binding:** React enforces a one-way data flow, making it clear how data is passed between components. This reduces unexpected side effects and makes debugging easier.
5. **Unidirectional Data Flow:** React's unidirectional data flow ensures that data flows in a single direction, simplifying state management and reducing the chance of data inconsistencies.
6. **JSX:** JSX allows developers to write UI components using a syntax that closely resembles HTML. This makes it more intuitive for developers to work with React.
7. **React Native**: React can be extended to build mobile applications using React Native. This allows developers to use their existing React knowledge to develop apps for iOS and Android.
8. **Server-Side Rendering (SSR):** React supports server-side rendering, which can improve initial page load times and provide better SEO performance.

**Render**

To render in web development means to convert HTML, CSS, and JavaScript code into a visual representation of a web page that a user can see in their browser. This process is done by the browser's rendering engine.

* The browser parses the HTML code to build a DOM tree. The DOM tree is a representation of the structure of the web page.
* The browser applies the CSS styles to the DOM tree to determine how the elements on the web page should be displayed.
* The browser renders the DOM tree to the screen. This is where the browser actually draws the elements on the web page to the screen.

**Components**

A component is a self-contained, reusable piece of code that defines how a part of a user interface should appear and behave. React applications are typically composed of multiple components that work together to create complex user interfaces.

* Components are the building blocks of React applications.
* They can be either functional or class-based.
* They encapsulate the UI and behaviour of specific parts of an application.
* Components can be reused and composed to create more complex user interfaces.
* Data can be passed to components via props, and components can maintain their own internal state when needed.
* Components can respond to events, handle user input, and update the UI based on changes in data or state.

**Differences between class components and functional component**

| **Aspect** | **Class Components** | **Functional Components** |
| --- | --- | --- |
| **Syntax** | Defined as JavaScript classes. | Defined as JavaScript functions. |
| **State Management** | Can hold and manage local state using **this.state** and **this.setState()**. | Stateless by default. Can use the **useState** hook to manage local state in modern React (React Hooks). |
| **Lifecycle Methods** | Have access to lifecycle methods like **componentDidMount**, **componentDidUpdate**, and **componentWillUnmount**. | Functional components can use lifecycle methods through React Hooks, such as **useEffect**. |
| **Complex Logic** | Suited for components with complex logic and state management needs. | Typically used for simple UI rendering with minimal logic. |
| **Readability** | Can be less concise and may have boilerplate code. | More concise and often considered easier to read and understand. |
| **Code Reusability** | Have the potential to share logic between lifecycle methods. | Encourage the use of custom hooks or higher-order components (HOCs) for code sharing. |
| **Performance** | May have a slight performance advantage for certain use cases due to optimization opportunities. | Generally perform well and are optimized in modern React. Performance differences are minimal in practice. |
| **State Initialization** | Requires constructor and **super** calls for state initialization. | State can be initialized directly using the **useState** hook. |
| **Binding Event Handlers** | Event handlers may need to be manually bound (e.g., using **.bind(this)** in the constructor). | No need for manual binding, as arrow functions or function components auto-bind **this**. |

**DOM**

The Document Object Model (DOM) is a way for a web browser to represent the structure, style, and content of a web page. It is an interface that allows programming languages, such as JavaScript, to manipulate the content, structure, and style of a website.

The DOM is a tree-like structure, where each element in the web page is represented as a node in the tree. The root node of the tree is the document object, which represents the entire web page. Other nodes in the tree represent different elements of the web page, such as the head element, body element, paragraph elements, and image elements.

1. **Structure**: The DOM sees a web page as a tree-like structure with different parts (like paragraphs, images, and links) as branches and leaves.
2. **Interaction**: It allows programs to change what's on a web page, like adding new text, images, or links, or responding to things like button clicks.
3. **Cross-Platform**: It's not tied to one programming language or device. You can use it with different languages and on various devices.
4. **Dynamic Web Pages**: It's what makes web pages interactive and lets them update without having to reload the whole page.
5. **Accessibility**: It helps make sure web content is usable by people with disabilities.

**Difference between actual DOM and virtual DOM**

The actual DOM, or real DOM, is the browser's representation of a web page's structure and content. It is a tree-like structure, where each element in the web page is represented as a node in the tree. The root node of the tree is the document object, which represents the entire web page. Other nodes in the tree represent different elements of the web page, such as the head element, body element, paragraph elements, and image elements.

The virtual DOM is a lightweight JavaScript representation of the real DOM. It is a copy of the real DOM that is kept in memory. When a change is made to the virtual DOM, the library that manages the virtual DOM (such as React) compares the virtual DOM to the real DOM and updates the real DOM only where necessary.

The main difference between the actual DOM and the virtual DOM is that the virtual DOM is much faster to update. This is because the virtual DOM is just a copy of the real DOM, and it does not need to update the browser's representation of the web page until it is necessary.

**Difference between single Page and multi-page application**

| **Aspect** | **Single-Page Application (SPA)** | **Multi-Page Application (MPA)** |
| --- | --- | --- |
| Loading Approach | Initial load, then dynamic content updates | Full page reloads for each view |
| User Experience | Fluid and responsive | Noticeable transitions between pages |
| Routing | Client-side routing | Traditional server-based navigation |
| Development Complexity | More complex, with client-side routing | Simpler development with separate pages |
| Server Interaction | Primarily through API requests | Server renders HTML for each page |
| SEO Challenges | SEO challenges without additional measures | More SEO-friendly by default with separate pages |
| Perceived Performance | Faster due to reduced full-page reloads | Slower due to full-page reloads |

**Top of Form**

**CSR vs SSR**

| **Aspect** | **Client-Side Rendering (CSR)** | **Server-Side Rendering (SSR)** |
| --- | --- | --- |
| **Rendering Location** | In CSR, rendering occurs in the browser. | In SSR, rendering occurs on the server. |
| **Initial Page Load** | Typically, faster because HTML is minimal and JavaScript is used to render the content. | May be slower because the server sends fully rendered HTML to the client. |
| **SEO Friendliness** | May require additional techniques (e.g., pre-rendering) for good SEO (Search Engine Optimization), as search engines may not wait for JavaScript execution. | SEO-friendly by default because fully rendered HTML is sent to the client. |
| **Performance Impact** | Slower initial load due to JavaScript processing for rendering. However, subsequent interactions can be faster with data fetching via APIs. | Faster initial load as the server sends pre-rendered HTML. Subsequent interactions may require additional API calls. |
| **Complexity** | Typically, more complex because it involves client-side routing, state management, and handling data fetching on the client. | Simpler in terms of client-side code as most rendering logic is handled on the server. |
| **Network Requests** | Requires additional network requests to fetch data (e.g., JSON) and render it on the client. | Fewer network requests for initial page load because HTML is pre-rendered. |
| **Time to Interactive (TTI)** | May have a slower TTI because it depends on JavaScript execution. | Faster TTI because much of the content is already rendered on the server. |
| **Caching** | Requires caching strategies for assets and API responses to improve performance. | Server-side caching can be more straightforward for full HTML pages. |

**Learn about es6 versions syntaxes**

1. **Arrow functions**

Arrow functions are particularly popular for defining anonymous functions and for simplifying the syntax when using functions as arguments in higher-order functions like map, filter, and reduce. They were introduced in ECMAScript 6 (ES6) and provide a shorter syntax compared to traditional function expressions.

**Key features of arrow functions:**

1. **Concise Syntax**: Arrow functions allow you to omit the **function** keyword and use a shorter syntax.
2. **Implicit Return**: If the function body contains a single expression, you can omit the curly braces **{}** and the **return** keyword. The result of the expression is automatically returned.
3. **No this Binding**: Arrow functions do not have their own **this** binding; they inherit the **this** value from the enclosing lexical context (usually the nearest non-arrow function). This behaviour can be advantageous in some cases, but it may not be suitable for all scenarios.
4. **Map**

**map()** is a built-in array method that allows you to iterate over the elements of an array and apply a given function to each element, creating a new array with the results of these function calls. The **map()** method does not modify the original array; instead, it returns a new array with the transformed values.

The function you provide as an argument to **map()** is called once for each element in the array, and the return value is used to create a new array. The original array remains unchanged.

1. **Destructuring**

Destructuring is a feature in JavaScript that allows you to extract values from objects or arrays and assign them to variables in a more concise and readable way. It simplifies the process of accessing and working with the properties or elements within data structures.

1. **Filter**

The **filter()** method is a built-in JavaScript array method that allows you to create a new array containing elements from an existing array that meet certain criteria. It iterates through the elements of the array and applies a callback function to each element. If the callback function returns **true** for an element, that element is included in the new array; otherwise, it is excluded.

1. **Reduce**

reduce() method allows you to iterate through the elements of an array and accumulate a single result by applying a given function to each element. It's often used for tasks like summing the values in an array, finding the maximum or minimum value, or aggregating data into a single value.

**Difference between JSX and HTML**

JSX is a JavaScript extension that simplifies the creation of user interfaces in React, while HTML is the standard markup language for structuring web content. The key differences lie in their usage, syntax, and support for dynamic behaviour.

* JSX is primarily a JavaScript extension used within React components, while HTML is a standalone markup language used to structure web content.
* JSX allows for embedding JavaScript variables and expressions within curly braces, making it suitable for dynamic content, while HTML relies on JavaScript for dynamic behaviour.
* JSX uses camelCase for attribute names and event handlers, whereas HTML uses lowercase attribute names.
* JSX provides more flexibility for handling events and dynamic content directly within its structure.
* HTML is natively supported by web browsers and doesn't require additional transpilation, whereas JSX code must be transpiled into regular JavaScript functions for browser compatibility.

Top of Form

**Class (basic- lifecycle methods)**

Lifecycle methods are special methods in React that are called at different stages in the lifecycle of a component. They allow you to perform specific actions at certain times, such as when a component is first created, when it is updated, or when it is removed from the DOM.

There are three main types of lifecycle methods:

* **Mounting lifecycle methods:** These methods are called when a component is first created and inserted into the DOM.
  + constructor() **:** This method is used to initialize the component's state and props.
  + getDerivedStateFromProps()**:** This method is used to calculate the component's state based on its props.
  + render(): This method is used to return the component's HTML markup.
  + componentDidMount()**:** This method is called after the component has been mounted and inserted into the DOM. It is a good place to fetch data and set up subscriptions.
* **Updating lifecycle methods:** These methods are called when a component's state or props change.
  + getDerivedStateFromProps()
  + shouldComponentUpdate()**:** This method is called before a component updates. It can be used to prevent unnecessary updates.
  + render()
  + getSnapshotBeforeUpdate()**:** This method is called before a component updates. It can be used to take a snapshot of the DOM before it is updated.
  + componentDidUpdate()**:** This method is called after a component has updated. It is a good place to update the DOM or perform other side effects.
* **Unmounting lifecycle methods:** These methods are called when a component is removed from the DOM.
  + componentWillUnmount()**:** This method is called before a component is removed from the DOM. It is a good place to clean up any resources that the component is using.

**Error Handling**

|  |  |
| --- | --- |
| **componentDidCatch(error, info)** | - Used to catch and handle errors that occur within the component's tree during rendering, lifecycle methods, or in constructors of child components. - Helps prevent the entire application from crashing due to an error in one component. |

**Dynamic rendering**

React only updates the real DOM when necessary to minimize performance overhead. Here are some examples of situations where React will update the real DOM:

* When a component's state changes.
* When a component's props change.
* When a component is mounted or unmounted.

**Here are some tips for optimizing re-renders in React:**

* **Use pure components:** Pure components are components that always return the same output for the same input. This means that pure components will not re-render unless their props or state change.
* **Use memoization:** Memoization is a technique that can be used to cache the results of expensive computations. This can help to reduce the number of times that a component needs to re-render.
* **Use lazy loading:** Lazy loading is a technique that can be used to delay the loading of components until they are actually needed. This can help to reduce the initial load time of your application and improve its performance.
* **Learn the concept of hooks**

1. **useState**
2. **useEffect**
3. **useRef**
4. **useHistory vs useNavigate**

* **Learn the concept of event handlers**

1. **onClick**
2. **onChange**

* **React router**

**Additional Topics**

**Fragment**

**Diffing Algorithm**

In React, the process of updating the user interface efficiently by comparing the Virtual DOM with the actual DOM and determining what needs to be changed is known as "**reconciliation**." The diffing algorithm is a crucial part of this reconciliation process, and it's responsible for identifying the differences between the Virtual DOM and the actual DOM.

React uses a tree diffing algorithm, to minimize the number of changes made to the actual DOM. When changes are made to a React component, a new Virtual DOM representation of the component's UI is created. React compares this new Virtual DOM with the previous one to identify differences. This comparison process is called **reconciliation**. React performs a top-down, depth-first traversal of both the old and new Virtual DOM trees. At each level of the tree, React identifies the differences between elements (e.g., components, elements, or text nodes) based on their type and properties.

**Memoization**

Memoization is a programming technique that caches the results of expensive function calls, and returns the cached results when the same inputs are encountered again. This can save a significant amount of time and improve performance, especially for functions that perform complex computations.

Memoization stores the results of computations in a cache. The cache is a data structure that stores key-value pairs. The keys are the inputs to the computation, and the values are the results of the computation.

When a computation is memoized, the function that performs the computation is first called with the given inputs. If the result of the computation is already stored in the cache, then the cached result is returned. Otherwise, the computation is performed and the result is stored in the cache.

**Higher order component**

HOC stands for Higher-Order Component. It is a design pattern in React that allows you to reuse component logic across multiple components. HOCs are functions that take a component as an argument and return a new component that wraps the original component.

HOCs are often used to add additional functionality to a component without modifying the component itself.

**Hooks:**

Hooks are functions provided by React that enable you to use state, lifecycle features, and more in functional components. They allow functional components to have the same capabilities as class components without the need for classes.

**Custom Hook**

Custom hooks are your own functions that use existing hooks to encapsulate and share complex logic between components, making your code more modular and reusable.

**useReducer**

useReducer is a React Hook that provides an alternative way to manage state in functional components, especially when dealing with complex state logic. It is often used as an alternative to the useState Hook when the state logic becomes more intricate.

The primary use case for useReducer is when you have state transitions that depend on the previous state and involve complex or nested state objects. It's also useful when you need to perform multiple state updates within a single action.

The **dispatch** function is used to dispatch actions to the reducer, triggering state updates.

**useRef**

In React useRef refers to a mechanism for storing data or values in a way that survives component re-renders without causing those re-renders. It allows you to preserve a piece of data or a reference to something while the component updates and re-renders.

When you use useRef, you create a mutable object that has a ‘**.current**’ property. This ‘**.current**’ property can hold any value, and its value persists across re-renders of the component, but changing it doesn't trigger a re-render.

useRef is mainly for mutable data and should not be used as a replacement for React's state management unless you specifically need to preserve values between renders without causing re-renders.

**useContext and Context API**

The Context API and the useContext hook are two different ways to use React Context. The Context API is a lower-level API that provides more flexibility, while the useContext hook is a higher-level hook that is easier to use.

**Context API**

The Context API is a way to create and share context objects. Context objects are plain objects that contain the data that you want to share. Once you have created a context object, you can use it to wrap any component that needs to access the context data. To do this, you use the Provider component. The Provider component takes a value prop, which is the context data that you want to share.

Any component that is wrapped by the Provider component can now access the context data using the Consumer component. The Consumer component takes a function as an argument, which is called with the context data.

Here is an example of how to use the Context API to share the current theme of an application:

// Create a context object

const ThemeContext = React.createContext();

// Wrap the App component with the Provider component

const App = () => {

const [theme, setTheme] = useState("light");

return (

<ThemeContext.Provider value={theme}>

<MyApp />

</ThemeContext.Provider>

);

};

// Use the Consumer component to get the current theme in the MyComponent component

const MyComponent = () => {

return (

<ThemeContext.Consumer>

{(theme) => (

<h1>Theme: {theme}</h1>

)}

</ThemeContext.Consumer>

);

};

**useContext hook**

The useContext hook is a higher-level hook that makes it easier to use React Context. The useContext hook takes a context object as an argument and returns the context data.

Here is an example of how to use the useContext hook to get the current theme of an application:

// Create a context object

const ThemeContext = React.createContext();

// Wrap the App component with the Provider component

const App = () => {

const [theme, setTheme] = useState("light");

return (

<ThemeContext.Provider value={theme}>

<MyApp />

</ThemeContext.Provider>

);

};

// Use the useContext hook to get the current theme in the MyComponent component

const MyComponent = () => {

const theme = useContext(ThemeContext);

return (

<h1>Theme: {theme}</h1>

);

};

**Proptypes**

PropTypes in React are a way to specify the types of props (short for properties) that a component should receive. They help catch and prevent bugs related to incorrect or missing prop values by defining the expected types for props in a component. Props are like inputs that you provide to a React component.

PropTypes are a set of rules you define to ensure those inputs (props) are of the right type.

**React.lazy**

React.lazy is a function in React that enables code-splitting for your components. Code-splitting is a technique used to split your JavaScript bundle into smaller files that are loaded on-demand. It helps reduce the initial load time of your application by only loading the code for a component when it's actually needed.

**vdom, useeffect, usestate, codecleaning, userefs, custom hooks,**

**Error boundaries**

**Optional chaining**

**Component life cycle method in detail**

**callback**

**library vs Framework**

**How to improve the performance of reactJs**

**useImperativeHandle**

**useLayoutEffect,**

**Child to Parent communication**

**Axios interceptors**

**Axios canceltoken**