**HTML**

**1. Difference between id and class**

An **id** is a unique identifier for a single element on a page, while a **class** can be applied to multiple elements. Think of an id as a person's Social Security number (unique), and a class as a person's occupation (many people can have the same one).

**2. HTML5 semantic elements**

HTML5 semantic elements like <header>, <nav>, <article>, and <footer> improve **SEO** and **accessibility**. They provide meaning and structure to the content, which helps search engines understand the context and relevance of a page. For accessibility, screen readers can navigate the page more effectively, providing a better experience for users with visual impairments.

**CSS**

**3. Box model**

The **CSS Box Model** is a fundamental concept that describes how elements are rendered. Every element is a rectangular box, with content, padding, border, and margin. **Content** is the actual element, **padding** is the space between the content and the border, **border** is the line surrounding the padding and content, and **margin** is the space outside the border.

**4. Centering a div using Flexbox**

To center a div both horizontally and vertically using Flexbox, you apply the following CSS to the parent container:

CSS

.container {

display: flex;

justify-content: center; /\* Horizontally centers content \*/

align-items: center; /\* Vertically centers content \*/

}

**5. Relative, absolute, and fixed positioning**

* **position: relative;**: The element is positioned relative to its normal position. Offsets like top, bottom, left, and right move the element from its original spot.
* **position: absolute;**: The element is positioned relative to its nearest positioned ancestor. If there is no positioned ancestor, it's relative to the initial containing block (the <html> element).
* **position: fixed;**: The element is positioned relative to the viewport. It stays in the same place even when the page is scrolled.

**6. Specificity in CSS**

**Specificity** is the algorithm browsers use to determine which CSS rule to apply to an element. It's calculated based on a hierarchy: inline styles > **ID selectors** > **class, attribute, pseudo-class** selectors > **element, pseudo-element** selectors. A higher specificity value means the rule will be prioritized.

**7. Media queries**

**Media queries** are a CSS feature that allows you to apply styles based on the characteristics of the device, such as screen size, resolution, and orientation. They are essential for creating responsive web designs. **Example:**

CSS

@media (max-width: 600px) {

body {

background-color: lightblue;

}

}

This example changes the background color of the body to lightblue when the viewport width is 600 pixels or less.

**8. Difference between em, rem, px, and % units**

* **px (pixels)**: An absolute unit. It's a fixed size and doesn't scale.
* **em**: A relative unit based on the font size of the parent element. For example, if a parent's font size is 16px, 1em is 16px.
* **rem (root em)**: A relative unit based on the font size of the root HTML element (<html>). This provides a more consistent scaling across the document, as it doesn't depend on the parent.
* **%**: A relative unit that is based on the size of the parent element.

**9. z-index**

The **z-index** property controls the vertical stacking order of positioned elements. An element with a higher z-index value will be displayed in front of an element with a lower value. It only works on elements with a position other than static.

**10. Pseudo-elements and pseudo-classes**

* **Pseudo-classes** select elements that are in a specific state, like a hovered link (:hover), a visited link (:visited), or the first child of a parent (:first-child).
* **Pseudo-elements** style specific parts of an element, like the first line of a paragraph (::first-line), the first letter (::first-letter), or content before/after an element (::before, ::after).

**JavaScript**

**11. Hoisting**

**Hoisting** is a JavaScript behavior where variable and function declarations are moved to the top of their containing scope during the compilation phase. This means you can use a variable or call a function before it has been declared in your code. However, only the declaration is hoisted, not the initialization.

**12. == vs ===**

* The == (loose equality) operator compares two values after converting them to a common type. It checks for value equality, not type equality.
* The === (strict equality) operator compares both the value and the type of the operands. This is generally preferred to avoid unexpected type coercion behavior.

**13. Closure**

A **closure** is a function that remembers and has access to its outer scope variables, even after the outer function has finished executing. **Example:**

JavaScript

function makeAdder(x) {

return function(y) {

return x + y;

};

}

const addFive = makeAdder(5);

console.log(addFive(2)); // Output: 7

Here, addFive is a closure that "remembers" the x variable from makeAdder.

**14. var, let, and const**

* **var**: Function-scoped. It can be re-declared and updated. It is hoisted.
* **let**: Block-scoped. It can be updated but not re-declared in the same scope. It is not hoisted in the same way as var.
* **const**: Block-scoped. It cannot be updated or re-declared. It must be initialized at the time of declaration.

**15. The Event Loop**

The **Event Loop** is a JavaScript mechanism that allows the execution of asynchronous code. It continuously checks the **call stack** and the **callback queue**. If the call stack is empty, it takes a callback from the queue and pushes it onto the stack to be executed.

**16. The this keyword**

The **this** keyword's value is determined by how a function is called.

* **Global context**: this refers to the global object (window in browsers, global in Node.js).
* **Object method**: this refers to the object itself.
* **Constructor**: this refers to the new object being created.
* **Explicit binding (call, apply, bind)**: this is explicitly set to a specific object.
* **Arrow functions**: this retains the value of the enclosing lexical context.

**17. Synchronous vs. Asynchronous code**

* **Synchronous code** is executed sequentially, one line at a time. The program must wait for one task to complete before moving to the next.
* **Asynchronous code** allows a program to continue its execution without waiting for a task to complete. This is useful for long-running operations like fetching data from a server.

**18. Promises and async/await**

* A **Promise** is an object representing the eventual completion or failure of an asynchronous operation. It has three states: pending, fulfilled, or rejected.
* **async/await** is syntactic sugar built on top of Promises. async functions always return a promise, and the await keyword pauses the function's execution until a Promise is resolved, making asynchronous code look synchronous and easier to read.

**19. Debouncing and Throttling**

* **Debouncing** is a technique that delays the execution of a function until after a certain amount of time has passed without the function being called again. This is useful for events like resizing or typing, where you only want to perform an action once the user has stopped.
* **Throttling** limits the number of times a function can be called over a specific period. It ensures the function executes at most once every X milliseconds. This is useful for things like scroll events.

**20. null vs undefined**

* **undefined** means a variable has been declared but has not been assigned a value.
* **null** is an assignment value that explicitly means "no value." It's a primitive value representing the intentional absence of any object value.

**React.js**

**21. React Hooks**

**React Hooks** are functions that let you "hook into" React state and lifecycle features from function components. They were introduced in React 16.8 to allow for the use of state and other React features without writing a class.

* **useState**: Adds state to a function component.
* **useEffect**: Performs side effects in a function component (e.g., data fetching, DOM manipulation).
* **useContext**: Accesses a React Context object.
* **useReducer**: An alternative to useState for more complex state logic.

**22. Controlled vs. Uncontrolled Components**

* A **controlled component** is a component where the form data is handled by React state. The state is the "single source of truth" for the input field.
* An **uncontrolled component** is where the form data is handled by the DOM itself. You access the form's value using a ref.

**23. Virtual DOM**

The **Virtual DOM (VDOM)** is a lightweight, in-memory representation of the actual DOM. When a component's state changes, React updates the VDOM first, then uses a "diffing" algorithm to compare the updated VDOM with the previous one. It then calculates the most efficient way to apply these changes to the real DOM, minimizing expensive updates.

**24. Props vs. State**

* **Props (short for properties)** are immutable and are passed from a parent component to a child component. They are used to pass data down the component tree.
* **State** is mutable and represents the internal data of a component. It is managed within the component itself and can change over time, triggering a re-render.

**25. Lifting State Up**

**Lifting state up** is a pattern in React where you move the state from a child component to its closest common ancestor. This is used when multiple child components need to share or update the same state, ensuring that the data is synchronized and consistent.

**26. Keys in React**

**Keys** are a special string attribute you need to include when creating lists of elements. React uses keys to identify which items have changed, been added, or been removed. This helps React's reconciliation algorithm efficiently update the UI and maintain the identity of each component instance.

**27. useEffect vs. useLayoutEffect**

* **useEffect** runs asynchronously after the browser has painted the screen. This is the more common hook and is safe to use for most side effects.
* **useLayoutEffect** runs synchronously after all DOM mutations but *before* the browser paints the screen. This is useful for reading DOM layout and making synchronous updates to the DOM. Using this hook can cause performance issues if not used correctly.

**28. Reconciliation and Re-rendering**

* **Reconciliation** is the process by which React compares the new virtual DOM tree with the old one to determine which parts of the actual DOM need to be updated. This "diffing" process makes updates efficient.
* **Re-rendering** is the process where React updates the user interface. A component re-renders when its state or props change, triggering the reconciliation process.

**29. Context API**

The **Context API** provides a way to pass data through the component tree without having to pass props down manually at every level. It's useful for "global" data, like user authentication status or theme settings, that many components need to access.

**30. Higher-Order Components (HOCs)**

A **Higher-Order Component** is a function that takes a component as an argument and returns a new component. HOCs are a pattern for reusing component logic. For example, you might use an HOC to share data fetching logic or to provide an authenticated wrapper around a component.

**Node.js**

**31. Node.js vs. Browser Environment**

* **Node.js** is a JavaScript runtime environment that runs on the server, providing access to the file system, network, and other OS features.
* The **browser environment** is a JavaScript runtime that runs on the client-side, providing access to the DOM, local storage, and other browser APIs. Node.js is not in the browser; it's a separate environment for server-side development.

**32. Middleware functions in Express.js**

**Middleware** are functions that have access to the request object (req), the response object (res), and the next middleware function in the application’s request-response cycle. They can execute code, modify the request and response objects, and end the cycle or pass control to the next function.

**33. Event-driven model in Node.js**

The **event-driven model** in Node.js is based on an event loop and callbacks. When an asynchronous operation is initiated, Node.js registers a callback and continues to the next task. When the operation completes, it emits an event, and the event loop executes the corresponding callback. This non-blocking I/O model is what makes Node.js highly efficient for concurrent operations.

**34. Benefits of using Express.js**

**Express.js** is a minimal and flexible web application framework for Node.js. Its benefits include:

* A rich set of features for building web and mobile applications.
* A robust routing system.
* Middleware support for handling requests.
* A simple and easy-to-use API.

**35. Handling errors in Node.js**

Errors in Node.js are typically handled using **callback functions with an error-first approach** or with **try...catch blocks** for synchronous code. For asynchronous code, Promises and async/await offer a cleaner way to handle errors with .catch() blocks.

**36. Role of package.json**

The **package.json** file is a manifest file for a Node.js project. It contains metadata about the project, such as its name, version, and author, but most importantly, it lists the project's **dependencies** and **scripts** for running common tasks (e.g., npm start, npm test).

**37. HTTP methods**

* **GET**: Retrieves data from a specified resource.
* **POST**: Submits data to be processed to a specified resource.
* **PUT**: Updates a resource or creates a new one with the provided data.
* **DELETE**: Deletes a specified resource.
* **PATCH**: Applies partial modifications to a resource.

**38. Streams in Node.js**

**Streams** are objects that allow you to read data from a source or write data to a destination in a continuous flow. This is more memory-efficient than reading a whole file into memory at once, especially for large files. They are used in file I/O, network communication, and more.

**39. require vs import**

* **require** is a **CommonJS** module system used in Node.js. It is synchronous and loads modules at runtime.
* **import** is part of the **ES6** (ECMAScript 2015) module system. It is asynchronous and typically loads modules at parse time. It is now widely supported in modern browsers and can be used in Node.js with a transpiler or by setting the module type in package.json.

**40. Managing environment variables**

**Environment variables** are used to store configuration settings outside of the source code. In Node.js, you can access them via process.env. Tools like dotenv are commonly used to load variables from a .env file into process.env during development.

**API & Database**

**41. SQL vs. NoSQL databases**

* **SQL (Relational)** databases are table-based with a predefined schema. They are good for complex queries and maintaining data integrity. (e.g., MySQL, PostgreSQL)
* **NoSQL (Non-relational)** databases are more flexible and can store data in various formats (document, key-value, graph, etc.). They are better for handling large volumes of unstructured data and scaling horizontally. (e.g., MongoDB, Cassandra)

**42. CRUD operations**

**CRUD** is an acronym for the four basic functions of persistent storage:

* **C**reate: POST in REST.
* **R**ead: GET in REST.
* **U**pdate: PUT or PATCH in REST.
* **D**elete: DELETE in REST.

**43. REST and its principles**

**REST** (Representational State Transfer) is an architectural style for designing networked applications. Key principles include:

* **Client-Server**: Separation of concerns.
* **Stateless**: Each request from a client to the server must contain all the information needed to understand the request.
* **Cacheable**: Responses can be cached to improve performance.
* **Uniform Interface**: A single, consistent way for clients and servers to interact.
* **Layered System**: Components can be deployed in a layered architecture.

**44. PUT vs PATCH**

* **PUT** is used to replace a resource entirely. If a part of the data is missing, it will be set to null or a default value.
* **PATCH** is used to apply a partial update to a resource. You only send the data you want to modify, and the rest of the resource remains unchanged.

**45. HTTP Status codes**

* **200 OK**: The request was successful.
* **404 Not Found**: The requested resource could not be found.
* **500 Internal Server Error**: The server encountered an unexpected condition that prevented it from fulfilling the request.

**46. Securing a REST API**

You can secure a REST API by using:

* **Authentication**: Verifying the identity of the user (e.g., username/password).
* **Authorization**: Granting permissions to an authenticated user (e.g., roles).
* **HTTPS**: Encrypting communication between client and server.
* **JWT (JSON Web Tokens)**: A common method for stateless authentication.
* **Rate Limiting**: Preventing abuse by limiting the number of requests a user can make in a given time.

**47. CORS**

**CORS** (Cross-Origin Resource Sharing) is a browser security mechanism that restricts a web page from making requests to a different domain than the one that served the web page. To handle it, the server must include specific HTTP headers, such as Access-Control-Allow-Origin, to explicitly allow requests from other origins.

**48. Indexing in databases**

**Indexing** is a database technique used to speed up data retrieval. An index is a data structure that allows the database to quickly locate specific rows in a table without scanning the entire table. It works similarly to an index in a book, providing a quick way to find information.

**49. JWT and authentication**

A **JWT (JSON Web Token)** is a compact, URL-safe means of representing claims to be transferred between two parties. For authentication, a server generates a JWT containing user information and a signature. The client stores this token and sends it with each subsequent request. The server then verifies the signature to ensure the token hasn't been tampered with.

**50. Structuring a RESTful API**

A **RESTful API** is typically structured using **resource-based URLs** and **HTTP methods** to perform actions.

* Use plural nouns for resources (e.g., /users, /products).
* Use HTTP methods (GET, POST, PUT, DELETE) to indicate the desired action on the resource.
* Use meaningful status codes for responses (200 OK, 404 Not Found).
* Use nested resources to show relationships (e.g., /users/123/posts).
* Use pagination, filtering, and sorting to manage large datasets.