# ← HTML QUESTIONS →

**Answer.1:** <!DOCTYPE html> is not a tag of html. It is an Information to the browser about what document type to expect.

**Answer.2:** Semantic tag in html provides meaningful and descriptive element to structure web content. They improve accessibility, enhance search engine optimization, simplify code maintenance, and future-proof websites. In sort, semantic tags make web pages easier to understand, navigate and interpret by both humans and machines.

**Answer.3:** 1) HTML Tags: HTML tags are the markup symbols or name enclosed in angle brackets(<>). They define the structure and behaviour of elements within an HTML document. Tags are used to enclose and delimit specific portions of content, indicating how browsers should interpret and render them. For example, <p>, <h1>, <div>, and <a> all are HTML tags.

2) HTML Elements: HTML elements consist of an opening tag, content, and a closing tag. They represent individual components or parts of a webpage. An HTML element is created by enclosing content within the appropriate HTML tags. For example, the HTML element for a paragraph would be <p>//content</p>, where <p> is a opening tag, “//content” is the content, and </p> is the closing tag. Elements can also be self-closing, like <img> or <br> where no closing tag is required.

**Answer.4:**GitHub Link → <https://github.com/AvishkarVichare/ineuron-placement-resume>

**Answer.5:** Here Link is not respondable..

**Answer.6:**

* Improved code,
* Element forms,
* Consistency,
* Support rich media elements, and
* Offline Application Cache.

**Answer.7:** GitHub Link → <https://github.com/AvishkarVichare/ineuron-placement-music-player>

**Answer.8:** The image tag is used to embed the image in an HTML document whereas the figure tag is used to semantically organize the content of an image in the HTML document. It is critical to note that these two elements are not interchangeable.

**Answer.9:** 1) HTML Tags: HTML tags are the markup symbols or name enclosed in angle brackets(<>). They define the structure and behaviour of elements within an HTML document. Tags are used to enclose and delimit specific portions of content, indicating how browsers should interpret and render them. For example, <p>, <h1>, <div>, and <a> all are HTML tags.

2) HTML Elements: HTML elements consist of an opening tag, content, and a closing tag. They represent individual components or parts of a webpage. An HTML element is created by enclosing content within the appropriate HTML tags. For example, the HTML element for a paragraph would be <p>//content</p>, where <p> is a opening tag, “//content” is the content, and </p> is the closing tag. Elements can also be self-closing, like <img> or <br> where no closing tag is required.

Example of global attributes:

1. Class: Specific one or more CSS class names to associate with an element, allowing for targeted styling or JavaScript interactions.
2. Id: Specific a unique identifier for an element, which can be used to reference or manipulate the element via CSS or JavaScript.
3. Style: Defines inline CSS styles for an element, allowing for custom styling directly applied to the element.
4. Title: Specific additional information about an element, often displayed as a tooltip when the user hovers over the element.

**Answer.10:** Here Link is not respondable.. But I’ve create my own table

GitHub Link → ​​<https://github.com/AvishkarVichare/ineuron-placement-table>

# ← CSS QUESTIONS →

**Answer.1:** The box model in CSS is a concept that defines how elements are structured and displayed on a webpage. It treats elements as rectangular boxes with properties that control their dimensions, spacing, and positioning.

The box model includes the content, padding, border, and margin of an element, which collectively determine its overall size and layout.

**Answer.2:.** There are some commonly types of selectors in CSS:

* Element selector,
* Class selector,
* Id selector,
* Child selector, and
* Attribute selector
* Advantages of CSS selector:
* Granular targeting,
* Reusability and modularity,
* Specificity and priority,
* Flexibility and maintainability, and
* Improved performance

**Answer.3:**

1) VW(Viewport Width): 1 VW is equal to 1% of the viewport’s width. For example, if the viewport width is 1000 pixels, 1 VW would be equal to 10 pixels (1000 \* 1 % = 10 pixels).

2) VH(Viewport Height): 1 VH is equal to 1% of the viewport’s height. For example, if the viewport height is 800 pixels, 1 VH would be equal to 8 pixels (800 \* 1 % = 8 pixels).

* In contrast, PX (pixels) is an absolute unit of measurement in CSS that represents a fixed size. It does not change based on the viewport size or any other factors. For example, 10 PX would always represent 10 Pixels, regardless of the viewport width or height.

**Answer.4:** 1) Inline: Inline element do not start on a new line and only occupy the space needed for their content. They cannot have width and height properties applied to them.

2) Inline Block: Inline-block elements flow with surrounding content like inline elements, but they can have width and height properties applied to them, and they allow for vertical alignment and margin/padding settings.

3) Block: Block elements start on a new line and occupy the full available width. They can have width and height properties applied to them, and they create a block-level box that separates content from the previous and next elements.

**Answer.5:** 1) Content Box: The content box is the default box-sizing property value in CSS. With this value, the width and height of an element are calculated are calculated by considering only the content area, excluding any padding, border, or margin.

2) Border-box: The border-box value for the box-sizing property includes the content, padding, and border within the specified width and height of an element. In other words, the width and height values include the total space occupied by content, padding, and border, with the margin being added separately.

**Answer.6(1):** z-index is a CSS property that determines the stacking order of elements on a webpage in the z-axis. It controls which elements appear in front of or behind other elements. Elements with a higher z-index value are positioned in front of elements with a lower value. This property is particularly useful when working with overlapping or layered elements, allowing developers to control the visibility and layering of different elements on the page.

**Answer.6(2):.** 1) Grid: CSS Grid is a two-dimensional layout system that allows you to create complex grid-based layouts. It divides a webpage into rows and columns, allowing precise control over the placement and alignment of elements. Grid provides powerful capabilities for creating responsive designs, positioning elements within the grid cells, and creating flexible layouts.

2) Flexbox (Flexible Box): Flexbox is a one-dimensional layout system that provides a flexible way to arrange elements within a container. It is primarily designed for arranging elements in a single row or column. Flexbox allows you to easily control the size, order, and alignment of elements, making it well-suited for creating responsive designs and handling dynamic layouts.

Key differences:

* Layout Orientation,
* Browser Support, and
* Number of Dimensions.

**Answer.7:** 1) Absolute Position: Positioned relative to its closest positioned ancestor (if any), otherwise relative to the initial containing block. Does not affect the position of other elements. For Example: <div style="position: absolute; top: 50px; left: 50px;">Absolute position</div>

2) Relative Position: Positioned relative to its normal position in the document flow. Other elements are not affected, and space is reserved for the element in its original position. For Example: <div style="position: relative; top: 20px; left: 20px;">Relative position</div>

3) Fixed Position: Positioned relative to the viewport (browser window). It remains fixed even when the page is scrolled. For Example: <div style="position: fixed; top: 0; left: 0;">Fixed position</div>

4) Sticky Position: Initially positioned according to the normal flow of the document, but behaves like fixed positioning within a specific scroll range. It becomes "sticky" and remains fixed until the scroll reaches a specified point. For Example: <div style="position: sticky; top: 50px;">Sticky position</div>

**Answer.8:** GitHub Link → <https://github.com/AvishkarVichare/ineuron-placement-periodic-table>

**Answer.9:**

GitHub Link → <https://github.com/AvishkarVichare/ineuron-placement-layout>

**Answer.10:** GitHub Link → <https://github.com/AvishkarVichare/ineuron-placement-responsive>

**Answer.11:**

GitHub Link → <https://github.com/AvishkarVichare/ineuron-placement-ineuron-website>

**Answer.12:** Pseudo class: Pseudo-classes are used to select and style elements based on specific states or conditions. They are preceded by a colon (:) and target elements that are in a particular state or meet certain criteria. For Examples: `:hover`, `:active`, `:focus`, `:first-child`, `:nth-child()`, etc.

Different From Pseudo Elements:

* Pseudo-classes select elements based on states or conditions, while pseudo-elements create virtual elements.
* Pseudo-classes are preceded by a colon (`:`), while pseudo-elements are preceded by a double colon (`::`).
* Pseudo-classes target elements in a specific state, such as `:hover` or `:focus`, while pseudo-elements target specific parts of an element's content, like `::before` or `::after`.
* Pseudo-classes are supported in all modern browsers, while pseudo-elements have good browser support but may require the double colon notation (`::`) for proper usage in some cases.

# ← JavaScript QUESTIONS →

**Answer.1:** In JavaScript, hoisting refers to the behavior of moving variable and function declarations to the top of their containing scope during the compilation phase, before the code is executed. This means that regardless of where variables and functions are declared in the code, they are conceptually moved to the top of their scope. However, it's important to note that hoisting only applies to declarations, not initializations or assignments. When a variable is hoisted, only the declaration is moved to the top, while the initialization or assignment remains in its original position.

For example:

```

console.log(x); // Output: undefined

var x = 10;

```

In the example above, the variable x is hoisted to the top of its scope (which in this case is the global scope). When the code is executed, the variable x exists, but its value is undefined because the initialization (x = 10) happens after the console.log statement.

**Answer.2:**In JavaScript, higher-order functions are functions that can take other functions as arguments or return functions as their results. They enable powerful functional programming paradigms and allow for more expressive and concise code. Some common higher-order functions in JavaScript include `map`, `forEach`, `filter`, `reduce`, and `sort`, among others.

1. .map(): The `map` method is used to iterate over an array and create a new array with the results of applying a provided function to each element of the original array. It returns a new array with the same length as the original array.

For example:

```

const numbers = [1, 2, 3, 4, 5];

const doubledNumbers = numbers.map((num) => num \* 2);

console.log(doubledNumbers); // Output: [2, 4, 6, 8, 10]

```

1. .forEach(): The `forEach` method is used to iterate over an array and execute a provided function once for each element in the array. It does not return a new array; instead, it is typically used when you want to perform an action for each element in the array without creating a new array.

For example:

```

const numbers = [1, 2, 3, 4, 5];

numbers.forEach((num) => console.log(num));

// Output:

// 1

// 2

// 3

// 4

// 5

```

**Answer.3:** 1) .call(): The call() method invokes a function with a specified this value and individual arguments passed in as comma-separated values.

For example:

```

const person = {

name: "John",

greet: function (message) {

console.log(`${message}, ${this.name}!`);

},

};

const anotherPerson = {

name: "Jane",

};

person.greet.call(anotherPerson, "Hello");

// Output: Hello, Jane!

```

In this example, call() is used to invoke the greet() method of the person object, but with the anotherPerson object as the this value. It allows us to borrow the greet() method from person and use it in the context of anotherPerson.

2) .apply(): The apply() method is similar to call(), but it accepts arguments as an array or an array-like object.

For example:

```

const numbers = [1, 2, 3, 4, 5];

const maxNumber = Math.max.apply(null, numbers);

console.log(maxNumber); // Output: 5

```

In this example, apply() is used to find the maximum value in the numbers array using the Math.max() function. The null value is passed as the this value (which is not used in this case), and the numbers array is passed as an array argument to apply().

3) .bind(): The bind() method creates a new function with a specified this value and any initial arguments. It doesn't immediately invoke the function but returns a new function that can be called later.

For example:

```

const person = {

name: "John",

greet: function (message) {

console.log(`${message}, ${this.name}!`);

},

};

const sayHello = person.greet.bind(person, "Hello");

sayHello();

// Output: Hello, John!

```

In this example, bind() is used to create a new function sayHello that is bound to the person object. The this value is set to person, and the initial argument "Hello" is passed to the greet() method. Later, when sayHello() is called, it executes the greet() method with the bound this value and the initial argument.

**Answer.4:** 1) Event bubbling: Event bubbling is the default behavior in JavaScript, where an event occurring on an inner element is first handled by the innermost element and then propagated to its parent elements in the DOM hierarchy, up to the outermost element. This means that the event is said to "bubble up" from the inner element to its ancestors.

For example:

```

<div id="outer">

<div id="inner">

<button id="button">Click Me</button>

</div>

</div>

<script>

const button = document.getElementById("button");

const inner = document.getElementById("inner");

const outer = document.getElementById("outer");

button.addEventListener("click", function () {

console.log("Button clicked");

});

inner.addEventListener("click", function () {

console.log("Inner div clicked");

});

outer.addEventListener("click", function () {

console.log("Outer div clicked");

});

</script>

```

In this example, if you click the button, the event starts at the button element, triggers the event handler for the button, then bubbles up to the inner div, triggering its event handler, and finally reaches the outer div, triggering its event handler. The output will be:

```

Button clicked

Inner div clicked

Outer div clicked

```

2) Event capturing: Event capturing is the opposite of event bubbling. In this mechanism, the event is first captured by the outermost element in the DOM hierarchy and then propagated inward to the innermost element. This is less commonly used than event bubbling.

For example:

```

<div id="outer">

<div id="inner">

<button id="button">Click Me</button>

</div>

</div>

<script>

const button = document.getElementById("button");

const inner = document.getElementById("inner");

const outer = document.getElementById("outer");

button.addEventListener("click", function () {

console.log("Button clicked");

}, false); // `false` indicates using event bubbling (default)

inner.addEventListener("click", function () {

console.log("Inner div clicked");

}, false); // `false` indicates using event bubbling (default)

outer.addEventListener("click", function () {

console.log("Outer div clicked");

}, true); // `true` indicates using event capturing

</script>

```

In this example, the event handlers are registered with event capturing by setting the third parameter of addEventListener() to true. Now, when you click the button, the event first captures at the outer div, triggering its event handler, then moves inward to the inner div, triggering its event handler, and finally reaches the button, triggering its event handler. The output will be:

```

Outer div clicked

Inner div clicked

Button clicked

```

**Answer.5:** Function currying is a technique in functional programming where a function with multiple arguments is transformed into a sequence of functions, each taking a single argument. Each function in the sequence returns a new function that expects the next argument until all the arguments have been provided, and finally, it returns the result.

For example:

```

function add(a) {

return function (b) {

return a + b;

};

}

const addFive = add(5);

console.log(addFive(3)); // Output: 8

```

In this example, the add() function takes an argument a and returns a new function that takes another argument b. The returned function adds a and b together and returns the result.

**Answer.6:** 1) Code snippet 1: In the global execution context, the statements are executed sequentially. The console.log('First') is executed first, printing "First" to the console. Then, the setTimeout() function is called, which schedules the execution of the arrow function () => console.log('Second') to be executed asynchronously after a minimal delay (0 milliseconds in this case). The execution continues to the next line and console.log('Third') is executed, printing "Third" to the console. The arrow function in the setTimeout() is added to the event queue to be executed later. As the event loop processes the event queue, the arrow function will be executed, printing "Second" to the console.

2) Code snippet 2: Similarly, in the global execution context, the statements are executed sequentially. The console.log('First') is executed first, printing "First" to the console. Then, the secondCall() function is defined. After that, the setTimeout() function is called with secondCall as the callback, which schedules the execution of secondCall after a delay of 2000 milliseconds (2 seconds). Next, another setTimeout() function is called with an arrow function () => console.log('Third'), which is scheduled to be executed asynchronously with a minimal delay (0 milliseconds). The execution continues to the next line, and console.log('Third') is executed, printing "Third" to the console. As time elapses, the first setTimeout triggers the execution of secondCall, printing "Second" to the console. Lastly, as the event loop processes the event queue, the arrow function from the second setTimeout is executed, printing "Third" to the console.

**Answer.7:**In JavaScript, promises are objects used for handling asynchronous operations. They represent the eventual completion of an asynchronous operation and allow you to attach callbacks to be executed when the operation is completed or an error occurs.

A promise can be in one of three different states:

1. Pending: The initial state of a promise. It means that the asynchronous operation associated with the promise has not yet been fulfilled or rejected.
2. Fulfilled: The state of a promise when the asynchronous operation is successfully completed. It means that the promised result is available, and the associated callback function `.then()` can be executed.
3. Rejected: The state of a promise when the asynchronous operation encounters an error or fails. It means that the promised result could not be obtained, and the associated error callback function `.catch()` can be executed.

For example:

```

function fetchData() {

return new Promise((resolve, reject) => {

setTimeout(() => {

const data = { name: 'John', age: 30 };

// Simulating successful completion

resolve(data);

// Simulating failure

// reject(new Error('Failed to fetch data'));

}, 2000);

});

}

fetchData()

.then((data) => {

console.log('Data fetched:', data);

})

.catch((error) => {

console.error('Error:', error.message);

});

```

In this example, the fetchData() function returns a new promise. Within the promise's executor function, a simulated asynchronous operation is performed using setTimeout() to delay the resolution of the promise by 2000 milliseconds (2 seconds).

If the operation is successful, resolve(data) is called, passing the fetched data as the resolved value. Otherwise, if an error occurs, reject(new Error('Failed to fetch data')) is called, rejecting the promise with an error.

The promise is then consumed using .then() to handle the fulfilled state, where the fetched data is logged to the console. If an error occurs during the promise execution, it is caught using .catch(), and the error message is logged to the console.

By creating and consuming promises, you can handle asynchronous operations more effectively, providing a structured approach for handling success and error cases.

**Answer.8:** The this keyword refers to the object on which a function is being executed or the object that the function belongs to. It allows accessing properties and methods within the current execution context. The value of this is determined dynamically based on how the function is invoked.

For example:

```

const person = {

name: "John",

age: 30,

greet: function () {

console.log(`Hello, my name is ${this.name}.`);

},

};

person.greet(); // Output: Hello, my name is John.

```

In this example, person is an object with properties name, age, and a method greet. When person.greet() is called, the this keyword inside the greet method refers to the person object itself. Thus, this.name accesses the name property of the person object, resulting in the output Hello, my name is John.

The value of this is determined dynamically and can change depending on the way a function is called:

1. Global Context,
2. Function Context, and
3. Constructor Context.

**Answer.9:** 1) Event loop: The event loop is a mechanism in JavaScript that ensures non-blocking and asynchronous behavior. It continuously checks the call stack and the callback queue for tasks. If the call stack is empty, it picks tasks from the callback queue and pushes them onto the call stack for execution. This allows JavaScript to handle asynchronous operations without blocking the main thread.

2) Call Stack: The call stack is a data structure that keeps track of function calls in a program. It follows the Last-In-First-Out (LIFO) principle, where the most recently invoked function is at the top of the stack. Each time a function is called, a new frame is added to the stack, and when a function completes, its frame is removed. This allows the program to keep track of which function is currently being executed.

3) Callback queue: The callback queue (also known as the task queue) is a queue that holds callback functions waiting to be executed. When an asynchronous operation completes, its associated callback function is placed in the callback queue. The event loop checks the callback queue and moves the callbacks to the call stack for execution when the call stack is empty.

4) Micro Task queue: The micro task queue (also known as the promise queue or microtask queue) is a special queue that holds micro tasks. Micro tasks have higher priority than regular tasks in the callback queue. Promises and certain APIs (like queueMicrotask() and MutationObserver) schedule their callbacks as micro tasks. The micro task queue is processed before the callback queue when the call stack is empty, allowing for more immediate execution of micro tasks.

**Answer.10:**Debouncing is a technique used in JavaScript to optimize performance by delaying the execution of a function until a certain period of inactivity has passed. It is commonly used to optimize event handlers that are triggered frequently, such as scroll events, resize events, or keystrokes. Debouncing ensures that the function is called only after a specified time has elapsed since the last invocation, reducing the number of function calls and improving efficiency.

For example:

```

<!-- Html File -->

<input type="text" id="search-input" placeholder="Search...">

<div id="search-results"></div>

// Debounce function

function debounce(func, delay) {

let timeoutId;

return function() {

clearTimeout(timeoutId);

timeoutId = setTimeout(func, delay);

}

}

// Function to simulate API call and display search results

function search() {

const searchInput = document.getElementById('search-input');

const searchResults = document.getElementById('search-results');

const query = searchInput.value;

// Simulating API call with search results

const results = simulateAPICall(query);

// Display search results

searchResults.innerHTML = '';

results.forEach(result => {

const resultElement = document.createElement('p');

resultElement.textContent = result;

searchResults.appendChild(resultElement);

});

}

// Simulated API call

function simulateAPICall(query) {

// Simulating delay

const delay = 500;

// Simulating search results

const results = [

'Result 1',

'Result 2',

'Result 3',

// ...

];

// Simulating filtering based on the query

const filteredResults = results.filter(result =>

result.toLowerCase().includes(query.toLowerCase())

);

// Simulating delay in API response

return new Promise(resolve => {

setTimeout(() => {

resolve(filteredResults);

}, delay);

});

}

// Apply debouncing to the search function

const debouncedSearch = debounce(search, 300);

// Add event listener to the search input

const searchInput = document.getElementById('search-input');

searchInput.addEventListener('input', debouncedSearch);

```

In this example, we have a search input field (< input>) and a search results container (< div>). We want to implement debouncing so that the search function is not triggered on every keystroke, but only after a brief pause when the user stops typing.

The debounce function is a utility function that takes a function (func) and a delay in milliseconds (delay). It returns a new function that wraps the original function and uses a timer to delay its execution. If the wrapped function is called within the delay period, the timer is reset, effectively debouncing the function.

The search function is the callback function that is debounced. It simulates an API call and displays the search results in the results container. In this example, we're using a simulated API call using the simulateAPICall function, which returns a promise with search results after a delay.

We apply the debouncing by creating a debounced version of the search function using debounce(search, 300). This debounced version is assigned to the debouncedSearch variable.We then add an event listener to the search input field (searchInput) and attach the debouncedSearch function as the callback. The debounced function is triggered only after a 300ms pause between keystrokes, preventing excessive API calls and improving performance.

By using debouncing, we optimize the search functionality by ensuring that the API call is made only after the user has finished typing or paused for a short period, reducing unnecessary API requests and improving the user experience.

**Answer.11:** Closures are an important concept in JavaScript that allows functions to retain access to variables from their outer (enclosing) lexical scope even after the outer function has finished executing. In simpler terms, a closure is a function bundled together with its surrounding state (variables) at the time of creation.A closure is created when a nested function references variables from its outer function, forming a "closure" over those variables. The inner function has access to its own local variables, the variables of its parent function, and any global variables.

Here are a few use cases where closures are commonly employed:

1. Data Privacy: Closures are often used to create private variables or encapsulated data within functions. By defining variables within an outer function and returning an inner function that references those variables, we can control access to the variables and prevent them from being modified from the outside. This concept is known as the module pattern and is widely used for creating reusable code components.
2. Iterators and Generators: Closures play a vital role in implementing iterators and generators in JavaScript. An iterator is an object that produces a sequence of values, while a generator is a special function that can pause and resume its execution. Closures help maintain the state of iteration and store references to variables within the iterator or generator function.

Closures provide powerful capabilities in JavaScript by allowing functions to remember and access variables from their lexical environment even after the enclosing function has finished executing. This enables a range of advanced programming techniques and design patterns, leading to more flexible, modular, and efficient code.

**Answer.12:**GitHub Link → <https://github.com/AvishkarVichare/ineuron-placement-blog->

# ← React QUESTIONS →

**Answer.1:** React is a popular JavaScript library for building user interfaces. It was developed by Facebook and has gained widespread adoption in the web development community. React allows developers to create reusable UI components and efficiently manage the state of those components.

Advantages:

* Virtual DOM,
* Component-Based Architecture,
* Unidirectional Data Flow,
* JSX Syntax,
* Large Ecosystem and Community, and
* Reusable Components

**Answer.2:** The Virtual DOM (Document Object Model) is a key concept in React. It is a lightweight representation of the actual DOM that React uses to efficiently update and render components.

Advantages:

* Efficient Updates,
* Batched Updates,
* Cross-Platform Rendering, and
* Developer-Friendly Abstraction.

**Answer.3:** The lifecycle of React components refers to the various stages a component goes through during its existence. These stages are divided into three main phases: Mounting, Updating, and Unmounting. Each phase consists of specific methods that can be overridden to execute custom logic at different points in the component's lifecycle.

**Answer.4:**1) Functional Components:

* Written as JavaScript functions.
* Use the useState and other hooks to manage state and lifecycle.
* No "this" keyword is used.
* Simpler and more concise syntax.
* No access to lifecycle methods (prior to React 16.8).
* Recommended when state and lifecycle management is not needed, or for simpler components.

2) Class Components:

* Written as ES6 classes that extend the React.Component class.
* Use the "this" keyword to access props, state, and lifecycle methods.
* Have access to lifecycle methods like componentDidMount and componentDidUpdate.
* Can have both state and props.
* Recommended when more complex state management and lifecycle control are required, or for integrating with existing class-based codebases.

**Answer.5:** Hooks are functions provided by React that allow developers to use state and other React features in functional components. They were introduced in React 16.8 as a way to write reusable and stateful logic without writing class components.

Hooks:

* useState,
* useEffect,
* useContext,
* useRef,
* useMemo, and
* useCallback.

You can't use a hook directly in a class component, but you can use a hook in a wrapped function component with a render prop to achieve this. Before going ahead with this, if you're able to convert your class component to a function component, prefer that.

**Answer.6:** What are the LifeCycle method and the advantages of it?

**Ans.** Lifecycle methods in React are methods that are invoked at different stages of a component's life. They provide hooks into the component's lifecycle events, such as mounting, updating, and unmounting. These methods can be overridden to execute custom logic and perform actions at specific points in the component's lifecycle.

Advantages:

* Controlling Component Behavior,
* Optimizing Performance,
* Handling Errors,
* Accessing External Resources, and
* Implementing Cleanup Tasks.

**Answer.7:** The useState hook is a built-in hook in React that allows functional components to manage state. It provides a way to declare and update state variables within functional components, enabling them to have dynamic and interactive behavior.

Advantages:

* Simplified State Management,
* Declarative Approach,
* Multiple State Variables,
* Immutable Updates,
* Functional Updates, and
* Works with React Hooks Ecosystem.

**Answer.8:** The useEffect hook is a built-in hook in React that enables functional components to perform side effects. Side effects refer to any code that needs to be executed outside the normal component rendering process, such as data fetching, subscriptions, or manipulating the DOM.

Advantages:

* Managing Side Effects,
* Lifecycle Control,
* Cleanup,
* Integration with Asynchronous Operations, and
* Custom Hooks and Reusability.

**Answer.9:** GitHub Link → <https://github.com/AvishkarVichare/ineuron-placement-context-api>

**Answer.10:** The useReducer hook is a built-in hook in React that allows state management through a reducer function, similar to how state is managed in Redux. It provides an alternative to the useState hook when dealing with complex state logic or when state transitions involve multiple sub-values or actions.

Advantages:

* Handling Complex State Logic,
* Centralized State and Actions,
* Predictable State Updates,
* Support for Middlewares and Devtools, and
* Component Reusability.

**Answer.11:** GitHub Link → <https://github.com/AvishkarVichare/ineuron-placement-todo>

**Answer.12:** GitHub Link → <https://github.com/AvishkarVichare/ineuron-placement-react-counter>

**Answer.13:** GitHub Link → <https://github.com/AvishkarVichare/ineuron-placement-react-calculator>

**Answer.14:** GitHub Link → <https://github.com/AvishkarVichare/ineuron-placement-tic-tac-toe.git>

**Answer.15:** Prop drilling refers to the process of passing props down through multiple layers of components in order to reach a deeply nested component that needs access to those props. It can occur when components that are not directly related need to share data or communicate with each other.

To avoid prop drilling, we can use a technique called "lifting state up" or use context in React.

**Answer.16:** GitHub Link → <https://github.com/AvishkarVichare/ineuron-placement-task-manager>

# ← Express QUESTIONS →

**Answer.1:** GitHub Link → <https://github.com/AvishkarVichare/ineuron-placement-server>

**Answer.2:** A middleware in Express is a function that has access to the request and response objects, as well as the next middleware function in the application's request-response cycle. It can perform tasks such as modifying request or response objects, terminating the request-response cycle, or calling the next middleware function.

GitHub Link → <https://github.com/AvishkarVichare/middleware>

**Answer.3:** GitHub Link → <https://github.com/AvishkarVichare/blog-backend>

**Answer.4:** 1) Authentication: Authentication is the process of verifying the identity of a user or entity. It ensures that the user is who they claim to be. In other words, authentication answers the question, "Who are you?" It is commonly achieved through the use of credentials such as usernames and passwords, biometric data (e.g., fingerprints or facial recognition), security tokens, or other authentication factors. Successful authentication grants access to the system or application.

2) Authorization: Authorization, on the other hand, is the process of granting or denying access to specific resources or functionalities based on the authenticated user's privileges or permissions. It determines what actions or operations a user is allowed to perform within the system. Authorization is typically based on roles, permissions, or access control lists (ACLs) defined by the system administrator or application developer. It answers the question, "What are you allowed to do?"

**Answer.5:** 1) Common JS: CommonJS is a module system used in Node.js for structuring and loading JavaScript modules. It follows a synchronous approach to module loading, where modules are loaded and executed synchronously. CommonJS modules use the require() function to import modules and the module.exports or exports object to define and export module contents.

2) EJS module: ESM (ECMAScript Modules) is the official module system introduced in ECMAScript 6 (ES6) for JavaScript. It provides a standardized way to define, import, and export modules. ESM modules use the import and export statements to handle module dependencies and exports.

Key differences:

* CommonJS modules are primarily used in Node.js environments, while ESM modules are native to modern browsers and are becoming more widely supported in Node.js as well.
* CommonJS modules have a synchronous module loading mechanism, whereas ESM modules support asynchronous module loading and allow dynamic imports.
* ESM modules have a more standardized syntax and are more capable of tree-shaking and static analysis for efficient bundling, while CommonJS modules have a simpler syntax but lack certain optimizations.

**Answer.6:** JWT (JSON Web Token) is an open standard for securely transmitting information between parties as a JSON object. It is commonly used for authentication and authorization purposes in web applications. A JWT consists of three parts: a header, a payload, and a signature. The header contains information about the token, such as the algorithm used for signing. The payload contains the claims or data associated with the token, such as user information or permissions. The signature is generated using a secret key and is used to verify the integrity of the token.

GitHub Link → <https://github.com/AvishkarVichare/ineuron-placement-jwt>

**Answer.7:** Before storing a user's password into a database, it is essential to apply appropriate security measures to protect the password and prevent unauthorized access. Here are two important steps to take with the user's password:

1. Hashing: The password should be hashed before storing it in the database. Hashing is a one-way process that transforms the password into an irreversible string of characters. By hashing the password, even if the database is compromised, the original password cannot be easily obtained. When a user logs in, their entered password is hashed using the same algorithm, and the resulting hash is compared to the stored hash. If they match, the password is considered valid.
2. Salting: To enhance the security of hashed passwords, a random salt should be added to each password before hashing. A salt is a random string of characters that is unique for each user. Salting prevents the use of precomputed rainbow tables or other precomputed attacks against hashed passwords. It ensures that even if two users have the same password, their hashed values will be different due to the unique salts.

By applying these two measures (hashing and salting), the original password is not stored in the database, and even if the database is compromised, it would be extremely difficult for an attacker to retrieve the actual password.

**Answer.8:** The event loop is a fundamental concept in Node.js that allows for non-blocking, asynchronous programming. It is responsible for handling and dispatching events and callbacks in an efficient and orderly manner.

**Answer.9:** GitHub Link →

Backend: <https://github.com/AvishkarVichare/ineuron-placement-full-stack-ecomm-backend>