HTML

**Answer.1 HTML**

<!DOCTYPE html> is the first line of code required in every HTML document. The DOCTYPE declaration is an instruction to the web browser about what version of HTML the page is written in. This ensures that the web page is parsed the same way by different web browsers.

**Answer.2 HTML**

Semantic tags in HTML refer to the use of specific HTML elements that carry meaning and describe the structure and purpose of the content they enclose. These elements provide contextual information to both browsers and developers, making it easier to understand and interpret the content of a web page.

<header>, <nav>, <main>, <section> are some example of semantic tags.

Semantic tags were introduced in HTML5 to address the problem of using non-descriptive or generic tags (such as <div> or <span>) to structure web documents. By using semantic tags, developers can provide more meaningful and self-explanatory markup, improving the accessibility, maintainability, and search engine optimization (SEO) of a website.

**Answer.3 HTML**

HTML Tags: HTML tags are the building blocks of HTML markup. They are used to define the structure and formatting of the content within an HTML document. Tags are enclosed within angle brackets (< >) and usually come in pairs: an opening tag and a closing tag. The opening tag indicates the beginning of an element, and the closing tag marks the end.

For example:

<p>This is a paragraph.</p>

In the above example, <p> is the opening tag, and </p> is the closing tag. Together, they define a paragraph element.

HTML Elements: HTML elements consist of both the opening and closing tags along with the content they enclose. They represent a complete unit within an HTML document. Elements are created by placing tags around the desired content. For example, in the paragraph element mentioned above, <p> is the opening tag, </p> is the closing tag, and "This is a paragraph." is the content of the element.

**Answer.4 HTML**

|  |  |
| --- | --- |
| github repo link | **https://github.com/GopalkrishaRao/Placement\_Assignment\_Gopalkrishna\_H\_R/tree/main/HTML/Q4-Resume** |
| **Deployed Pjoect link** | [**https://html-resume-gkrao-aaa94b.netlify.app/**](https://html-resume-gkrao-aaa94b.netlify.app/) |

**Answer.5 HTML**

|  |  |
| --- | --- |
| github repo link | **https://github.com/GopalkrishaRao/Placement\_Assignment\_Gopalkrishna\_H\_R/tree/main/HTML/Q5-Image** |
| **Deployed Pjoect link** | **https://assignment-q5-1fa481.netlify.app/** |

**Answer.6 HTML**

HTML5 introduced several advancements and new features over its previous versions, offering numerous advantages for web development. Some of the key advantages of HTML5 are:

* Improved Semantics: HTML5 introduced a set of semantic elements (e.g., <header>, <nav>, <article>) that provide a clearer and more meaningful structure to web pages. This helps search engines better understand the content and improves accessibility.
* Native Multimedia Support: HTML5 introduced native support for audio and video elements (<audio> and <video>). This eliminates the need for third-party plugins like Flash and enables developers to embed multimedia content directly into web pages.
* Canvas and SVG: HTML5 introduced the <canvas> element, which allows for dynamic and interactive rendering of graphics and animations using JavaScript. Additionally, HTML5 brought support for Scalable Vector Graphics (SVG), which enables the use of vector-based graphics that scale smoothly across different screen sizes.
* Offline Capabilities: HTML5 introduced the Application Cache (<appcache>) and Local Storage (localStorage) features, which enable web applications to work offline or in low-connectivity situations. Developers can cache resources and store data locally, enhancing the user experience and allowing offline functionality.
* Enhanced Forms: HTML5 introduced several new input types (<input type="date">, <input type="email">, <input type="range">, etc.) and form validation features. This simplifies form handling, improves user experience, and reduces the need for custom JavaScript validations.
* Responsive Design: HTML5 provides better support for responsive web design, allowing developers to create fluid and adaptive layouts that adjust to different screen sizes and devices. This is achieved through features like media queries and new layout elements such as <section> and <article>.
* Improved Performance: HTML5 includes various performance optimizations, such as the ability to load scripts asynchronously (async and defer attributes), which improves page loading speed. Additionally, HTML5 introduced the Web Workers API, enabling multi-threading and background processing for complex tasks.
* Geo-location: HTML5 introduced the Geolocation API, which allows web applications to retrieve the user's location information (with user consent). This feature enables location-based services and enhances the functionality of location-aware web applications.
* Cross-platform Compatibility: HTML5 is designed to work consistently across different platforms and devices, promoting cross-platform compatibility. It reduces the need for platform-specific coding and ensures a more unified experience for users.

**Answer.7 HTML**

|  |  |
| --- | --- |
| github repo link | **https://github.com/GopalkrishaRao/Placement\_Assignment\_Gopalkrishna\_H\_R/tree/main/HTML/Q7-MusicPlayer** |
| **Deployed Pjoect link** | **https://html-music-player-61936a.netlify.app/** |

**Answer.8 HTML**

<img> tag is used specifically for displaying images, while the <figure> tag is used to group self-contained content, such as an image, and its associated caption. The <figure> tag provides additional semantic meaning and aids in the representation and understanding of the relationship between the content and its description.

**Answer.9 HTML**

* HTML Tags: HTML tags are the building blocks of HTML documents. They define the structure and meaning of the content within an HTML document. Tags are enclosed within angle brackets (< >) and usually come in pairs: an opening tag and a closing tag.
* HTML Attributes: HTML attributes provide additional information about an HTML element. They are used within the opening tag of an element and modify the element's behavior, appearance, or define specific characteristics. Attributes consist of a name and a value and are placed inside the opening tag of an element.
* Global attributes are a set of attributes that can be used on most HTML elements. They provide common functionality and are not specific to any particular element. class, id, title etc are some examples of html global attributes.

**Answer.10 HTML**

|  |  |
| --- | --- |
| github repo link | **https://github.com/GopalkrishaRao/Placement\_Assignment\_Gopalkrishna\_H\_R/tree/main/HTML/Q10-Table** |
| **Deployed Pjoect link** | **https://assignment-q10-table-645291.netlify.app/** |

**CSS**

**Answer.1 CSS**

The CSS box model is essentially a box that wraps around every HTML element. It consists of: margins, borders, padding, and the actual content.

**Answer.2 CSS**

Following are the different types of selectors in CSS:

* Element selectors: These selectors target elements based on their HTML tag names.
* Class selectors: Class selectors target elements based on their class attribute. They are denoted by a dot (.) followed by the class name.
* ID selectors: ID selectors target elements based on their unique id attribute. They are denoted by a hash (#) followed by the ID name
* Attribute selectors: Attribute selectors target elements based on their attribute values.
* Pseudo-classes and pseudo-elements: Pseudo-classes select elements based on their state or position in the document tree. Examples include :hover (selects an element when the mouse hovers over it) Pseudo-elements target specific parts of an element, such as ::before (inserts content before an element) and ::after (inserts content after an element).

Advantages of different types of selectors are:

* Flexibility: CSS selectors provide different levels of granularity, allowing you to target specific elements or groups of elements based on your needs.
* Reusability: Class selectors can be applied to multiple elements, enabling consistent styling throughout a website.
* Specificity: ID selectors have high specificity, making them useful for targeting specific elements with precision.
* Dynamic styling: Pseudo-classes and pseudo-elements allow you to apply styles based on element states or insert content dynamically.

**Answer.3 CSS**

In CSS, "vw" and "vh" are relative length units that represent a percentage of the viewport's width and height, respectively. These units allow developer to size elements based on the dimensions of the viewport, providing a responsive and flexible design approach.

Pixels (px) are the absolute unit of measurement commonly used in CSS. One pixel represents the smallest unit of screen display and has a fixed size. The main difference between "vw"/"vh" and "px" is their behavior in relation to the viewport dimensions. While pixels provide a fixed size, "vw" and "vh" units offer responsiveness by scaling with the viewport.

**Answer.4 CSS**

Three commonly used values for the display property in CSS are "inline", "inline-block", and "block".

|  |  |  |
| --- | --- | --- |
| Inline | Inline-block | block |
|  |  |  |
| Elements with the inline display property do not start on a new line and only occupy the space necessary for their content. | Elements with the display property set to "inline-block" share characteristics of both inline and block elements. They flow inline like text but also allow width and height properties to be applied. Inline-block elements start on the same line as other inline elements, but they can have margins, padding, and dimensions | Elements with the display property set to "block" start on a new line and occupy the full width available within their parent container. |
| <span>, <a>, and <strong> are some of html tags which has default inline property | Examples of inline-block elements are <img>, <button>, and <input>. | Examples of block elements are <div>, <p>, <h1> to <h6>, and <ul>. |

**Answer.5 CSS**

In CSS, the box-sizing property used to control how the width and height of an element are calculated, affecting the total size of the box model. The two main values for the box-sizing property are "content-box" and "border-box".

1. Content-box:

The default value for the box-sizing property is "content-box". With this value, the specified width and height of an element only include the content area. It does not include the padding or border. The padding and border are added to the specified width and height, increasing the total dimensions of the box. In other words, the content box size is independent of the padding and border.

For example, if you set an element's width to 200px and add 10px padding and a 2px border, the total width of the element will be 224px (200px content width + 10px padding on the left + 10px padding on the right + 2px border on the left + 2px border on the right).

1. Border-box:

When we set the box-sizing property to "border-box", the specified width and height of an element includes both the content, padding, and border. The padding and border are included in the specified dimensions, and they do not increase the total width and height of the box. In other words, the border box size is determined by the specified width and height, and the padding and border are included within that space.

Continuing with the previous example, if we set an element's width to 200px and set the box-sizing property to "border-box", the element's total width will remain 200px, and the padding and border will be contained within that width. The browser automatically adjusts the content width to accommodate the padding and border.

**Answer.6 CSS**

In CSS, the z-index property is used to control the stacking order of elements on a web page along the z-axis, which represents the depth or elevation of elements in the three-dimensional space. The z-index property only works on positioned elements (elements with a position value other than "static," such as "relative," "absolute," or "fixed").

The z-index property accepts integer values (positive, negative, or zero) to assign stacking levels. A higher positive value increases the element's stacking level, while a negative value decreases it.

By using the z-index property strategically, one can control the visual layering and stacking order of elements on your web page, allowing you to position elements in front of or behind other elements based on their stacking levels within their respective stacking contexts.

**Answer.6 CSS**

Grid and Flex are two CSS layout systems that are used to create responsive and flexible web page layouts. While they have some similarities, they serve different purposes and have distinct characteristics.

1. CSS Grid:

CSS Grid is a two-dimensional layout system that divides a webpage into rows and columns. It provides a grid-based structure that allows user to precisely position and align elements within the grid cells. Key features of CSS Grid include:

* Two-dimensional layout: CSS Grid allows you to define both rows and columns, giving you control over the placement and sizing of elements in both directions.
* Grid lines and tracks: Grid lines are the horizontal and vertical lines that define the boundaries of the grid cells. Grid tracks are the spaces between the grid lines, which can be set to specific sizes or auto to adjust based on content.
* Explicit positioning: With CSS Grid, you can explicitly position elements anywhere in the grid, allowing for precise control over their placement. You can define the starting and ending grid lines for each element.
* Grid alignment: CSS Grid provides various alignment properties to control the positioning of elements within the grid cells, such as aligning them vertically or horizontally.

1. CSS Flexbox:

CSS Flexbox is a one-dimensional layout system that operates in a single direction (either horizontally or vertically) along a flex container and its flex items. Flexbox is designed for creating flexible and dynamic layouts. Key features of CSS Flexbox include:

* One-dimensional layout: Flexbox focuses on arranging elements in a single row or column, depending on the flex container's direction.
* Flex container and flex items: The parent element becomes a flex container by setting the display property to "flex". Its child elements become flex items and flow within the flex container.
* Flexibility and auto-sizing: Flex items can dynamically adjust their width or height based on available space, accommodating different screen sizes and content lengths.
* Flex alignment: Flexbox provides alignment properties to control how flex items are aligned within the flex container, such as justifying their distribution along the main axis or aligning them vertically along the cross axis.

Differences between Grid and Flex are:

* Layout dimension: Grid is a two-dimensional layout system, while Flexbox is a one-dimensional layout system.
* Control over layout: Grid provides more fine-grained control and positioning of elements in both rows and columns. Flexbox focuses on flexible and dynamic layouts, allowing elements to expand and shrink based on available space.
* Layout complexity: Grid is ideal for complex layouts that require precise control over the positioning and alignment of elements. Flexbox is often used for simpler layouts or within Grid itself to control the behavior of items within a grid cell.
* Direction: Grid can handle elements arranged in both horizontal and vertical directions. Flexbox focuses on either horizontal (row) or vertical (column) layouts.

**Answer.7 CSS**

Position property in CSS helps to design attractive interfaces with ease and helps in positioning the elements.

There are five different position values: Static, Relative, Absolute, Fixed, Sticky

1. Relative position:

Syntax - position: relative;

Elements with relative positions are positioned relative to their normal position. Properties like the top, bottom, right, and left are used to adjust from their normal position. Other contents or elements will not be adjusted to fill or fit into the space left by the element.

1. Absolute position:

Syntax - position: absolute;

In absolute position, the elements are positioned relative to their ancestor or parent container. If there is no parent container then it takes the body of the page as its parent. In absolute position, the other elements surrounding it ignores the space occupied by the element with absolute position and the element is removed from the document flow in contrast to relative position.

1. Sticky position:

Syntax - position: sticky;

An element with a sticky position is positioned based on the scroll position. They shift between relative and fixed values based on the scroll. Element with sticky position is positioned relative until a given offset position is met in the viewport and then it gets fixed in place like position: fixed.

1. Fixed position:

Syntax - position: fixed;

Elements with the fixed position are positioned relative to the viewport and they do not weave when scrolled up or down. The top, right, bottom, and left properties are used to position the element.

**Answer.8 CSS**

|  |  |
| --- | --- |
| github repo link | **https://github.com/GopalkrishaRao/Placement\_Assignment\_Gopalkrishna\_H\_R/tree/main/CSS/Q8-Periodic%20table** |
| **Deployed Pjoect link** | [**https://css-periodic-table-8588d.netlify.app/**](https://css-periodic-table-8588d.netlify.app/) |

**Answer.9 CSS**

|  |  |
| --- | --- |
| github repo link | **https://github.com/GopalkrishaRao/Placement\_Assignment\_Gopalkrishna\_H\_R/tree/main/CSS/Q9-image** |
| **Deployed Pjoect link** | [**https://assignmetn-css-q9-daee3c.netlify.app/**](https://assignmetn-css-q9-daee3c.netlify.app/) |

**Answer.10 CSS**

|  |  |
| --- | --- |
| github repo link | **https://github.com/GopalkrishaRao/Placement\_Assignment\_Gopalkrishna\_H\_R/tree/main/CSS/Q10-Responsive%20Layout** |
| **Deployed Pjoect link** | **https://assignment-css-responsive-lay-e5a2c2a.netlify.app/** |

**Answer.11 CSS**

|  |  |
| --- | --- |
| github repo link | **https://github.com/GopalkrishaRao/Placement\_Assignment\_Gopalkrishna\_H\_R/tree/main/CSS/Q11-IneuronHomePage** |
| **Deployed Pjoect link** | **https://assignment-ineruon-home-page-4c397b.netlify.app/** |

**Answer.12 CSS**

Pseudo-classes select elements based on their state or a specific condition. They are denoted with a colon (:) followed by the pseudo-class name. Some commonly used pseudo-classes include:hover, :active, :focus, :nth-chind(n). Pseudo-classes are added to existing selectors to refine the selection based on the element's state or characteristics. They provide interactivity and dynamic styling based on user actions or element properties.

Pseudo-elements, on the other hand, target specific parts of an element to style or insert content before or after the element. They are denoted with a double colon (::) followed by the pseudo-element name. Some commonly used pseudo-elements include ::before, ::after, ::first-line, ::first-letter

**JavaScript**

**Answer.1 JavaScript**

Hoisting is a behavior in JavaScript where variable and function declarations are moved to the top of their respective scopes during the compilation phase. This means that regardless of where variables and functions are declared in the code, they are treated as if they are declared at the top of their scope . However, it's important to note that only the declarations are hoisted, not the initializations or assignments.

Hoisting occurs due to the way JavaScript's execution context and the creation of variable and function declarations are handled. Here are a few key points to understand about hoisting:

1. Variable Hoisting:

Variable declarations (using var) are hoisted to the top of their scope, which can be the global scope or a function scope. However, only the declarations themselves are hoisted, not their values or assignments. If you try to access a variable before its declaration, it will exist but have the value undefined. For example:

**console.log(x); // undefined**

**var x = 10;**

1. Function Hoisting:

Function declarations are also hoisted to the top of their scope. This means that you can invoke a function before its declaration in the code. For example:

**sayHello(); // "Hello"**

**function sayHello() {**

**console.log("Hello");**

**}**

1. Function Expressions:

Function expressions, where a function is assigned to a variable, are not hoisted. Only the variable declaration is hoisted, not the function assignment. For example:

**sayHello(); // Error: sayHello is not a function**

**var sayHello = function() {**

**console.log("Hello");**

**};**

**Answer.2 JavaScript**

In JavaScript, higher-order functions are functions that can take other functions as arguments or return functions as results. .map() and .forEach() are some examples of higher-order functions.

Difference between .map() and .forEach() are

|  |  |  |
| --- | --- | --- |
|  | .map() | .forEach() |
| Return Value | returns a new array with the transformed values | performs an action on each element without returning any values |
| Usage | used to transform each element of an array and create a new array with the transformed values | used to iterate over an array and perform an action on each element, without creating a new array. |
| Immutability | it ensures immutability by not modifying the original array. | modify the original array directly within the callback function. |
| Method Chaining | returns a new array so, can chain other array methods like filter() or reduce() after it | does not return a value, so it cannot be chained with other array methods. |

**Answer.3 JavaScript**

In JavaScript, the methods .call(), .apply(), and .bind() are used to manipulate the execution context and parameters of a function. They allow you to explicitly set the value of this and pass arguments to the function. Here's an explanation of each method along with an example:

1. .call():

The .call() method is used to invoke a function with a specified this value and individual arguments passed as separate arguments. It takes the context (the value to be set as this) as the first argument, followed by individual arguments. For example:

**const person = {**

**name: 'John',**

**sayHello: function() {**

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

**}**

**};**

**const anotherPerson = {**

**name: 'Alice'**

**};**

**person.sayHello.call(anotherPerson); // Output: Hello, Alice!**

In this example, we have an object person with a method sayHello(). By using .call(), we invoke sayHello() with the context of anotherPerson, overriding the default this value of person to anotherPerson.

1. .apply():

The .apply() method is similar to .call(), but it accepts arguments as an array or an array-like object instead of separate arguments. The first argument is still the context (the value to be set as this), and the second argument is an array or an array-like object containing the arguments. For example:

**function sayHello(message, punctuation) {**

**console.log(message + ', ' + this.name + punctuation);**

**}**

**const person = {**

**name: 'John'**

**};**

**const args = ['Hello', '!'];**

**sayHello.apply(person, args); // Output: Hello, John!**

Here, we have a function sayHello() that accepts two arguments. By using .apply(), we pass person as the context and an array args containing the arguments to be passed to sayHello().

1. .bind():

The .bind() method returns a new function with a specified this value and any initial arguments. It allows you to create a function that, when invoked, will have a specific context and pre-set arguments. For example:

**const person = {**

**name: 'John',**

**sayHello: function() {**

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

**}**

};

**const greet = person.sayHello.bind(person);**

**greet(); // Output: Hello, John!**

Key differences between .call(), .apply(), and .bind():

* .call() and .apply() invoke the function immediately, while .bind() returns a new function that can be invoked later.
* .call() and .apply() accept the function arguments as separate arguments or an array-like object, respectively, while .bind() accepts the arguments when creating the new function.
* .call() and .bind() can set the this value and pass arguments, while .apply() can only set the this value and pass arguments as an array or array-like object.

**Answer.4 JavaScript**

Event bubbling and event capturing are two different mechanisms in JavaScript for handling events when they occur on nested elements within the DOM (Document Object Model) hierarchy. They define the order in which event handlers are executed during the event propagation process. Let's explain each mechanism with an example:

1. Event Bubbling:

In event bubbling, when an event is triggered on an element, it first fires the event handlers on the innermost element and then propagates upward through its ancestors, triggering their event handlers in sequence. The event bubbles up from the innermost element to the outermost element in the DOM hierarchy.

Here's an example:

**<div id="outer">**

**<div id="inner">**

**<button id="button">Click me</button>**

**</div>**

**</div>**

**<script>**

**const outer = document.getElementById('outer');**

**const inner = document.getElementById('inner');**

**const button = document.getElementById('button');**

**button.addEventListener('click', () => {**

**console.log('Button clicked!');**

**});**

**inner.addEventListener('click', () => {**

**console.log('Inner div clicked!');**

**});**

**outer.addEventListener('click', () => {**

**console.log('Outer div clicked!');**

**});**

**</script>**

When you click the button, the event handlers will be executed in the following order: Inner div clicked!>Outer div clicked!>Button clicked!

As you can see, the event starts at the innermost element (button) and bubbles up to the outermost element (outer), triggering the event handlers along the way.

1. Event Capturing:

In event capturing, the event is first captured by the outermost element and then propagates downward through its descendants, triggering their event handlers in sequence. The event travels from the outermost element to the innermost element in the DOM hierarchy.

Here's an example:

**button.addEventListener('click', () => {**

**console.log('Button clicked!');**

**}, { capture: true });**

**inner.addEventListener('click', () => {**

**console.log('Inner div clicked!');**

**}, { capture: true });**

**outer.addEventListener('click', () => {**

**console.log('Outer div clicked!');**

**}, { capture: true });**

With the { capture: true } option passed to each addEventListener() call, the event handlers will be executed in the following order when clicking the button:

Outer div clicked! > Inner div clicked! > Button clicked!

In event capturing, the event starts at the outermost element (outer) and captures the event as it propagates down to the innermost element (button), triggering the event handlers in that order.

It's important to note that event capturing and event bubbling are part of the event propagation process, and by default, event handling in JavaScript follows the event bubbling mechanism. You can use the addEventListener() method with the { capture: true } option to enable event capturing explicitly.

Understanding event bubbling and event capturing is essential for managing event delegation and handling events on nested elements efficiently. It allows you to control the order in which event handlers are executed based on your requirements.

**Answer.5 JavaScript**

Function currying is a technique in JavaScript where a function with multiple parameters is transformed into a sequence of functions, each taking a single parameter. The resulting sequence of functions can be called one by one, each with a single argument, until all the arguments are supplied and the final result is returned. It allows for partial application of arguments, which can be useful for creating reusable and specialized functions. Here's an example to illustrate function currying:

**function multiply(a) {**

**return function(b) {**

**return a \* b;**

**};**

**}**

**const multiplyByTwo = multiply(2);**

**console.log(multiplyByTwo(4)); // Output: 8**

**console.log(multiplyByTwo(6)); // Output: 12**

In this example, we have a multiply function that takes one parameter a. Inside the function, it returns another function that takes a parameter b and returns the multiplication of a and b.

By calling multiply(2), we create a new function multiplyByTwo that is specialized for multiplying by 2. When we call multiplyByTwo(4), it multiplies 2 by 4 and returns 8. Similarly, calling multiplyByTwo(6) returns 12.

The function currying technique allows us to create specialized functions by partially applying arguments. In this case, we created a reusable function multiplyByTwo that can be used to multiply any number by 2 without explicitly providing the 2 argument every time.

Currying can be helpful in scenarios where you want to create specialized versions of a function with certain arguments pre-set, allowing for cleaner and more concise code. It enables you to create function factories that generate new functions based on a common pattern.

**Answer.6 JavaScript**

**Code Snippet 1**

**console.log('First');**

**setTimeout(() => console.log('Second'), 0);**

**console.log('Third');**

|  |  |
| --- | --- |
| **Memory** | **Code** |
| **First**  **Third** | |  |  | | --- | --- | | **Memory** | **Code** | | **setTimeout(() => console.log('Second'), 0);** |  | |

The Output of above code will be:

**First**

**Third**

**Secoend**

|  |  |
| --- | --- |
| **Memory** | **Code** |
| **First**  **Third**  **Secoend** | |  |  | | --- | --- | | **Memory** | **Code** | | **setTimeout(() => console.log('Second'), 0);** |  | |

The code snippet starts in the global execution context. ‘First’ is printed first and as ‘setTimeout(‘) function is asynchronous as code moves to next statement and it is schedules callback for later execution allowing to print ‘Third ‘ later at last Secoend is printed.

**Code Snippet 2**

**console.log('First');**

**function secondCall() {**

**console.log('Second');**

**}**

**setTimeout(secondCall, 2000);**

**setTimeout(() => console.log('Third'), 0);**

**console.log('Third');**

The Output of above code will be:

**First**

**Third**

**Third**

**Second**

|  |  |
| --- | --- |
| **Memory** | **Code** |
| **First**  **function secondCall()**  **Third** | |  |  | | --- | --- | | **Memory** | **Code** | | **setTimeout(secondCall, 2000);**  **setTimeout(() => console.log('Third'), 0);** |  | |

|  |  |
| --- | --- |
| **Memory** | **Code** |
| **First**  **function secondCall()**  **Third**  **Third** | |  |  | | --- | --- | | **Memory** | **Code** | | **setTimeout(secondCall, 2000);** |  | |

|  |  |
| --- | --- |
| **Memory** | **Code** |
| **First**  **function secondCall()**  **Third**  **Third**  **Secoend** |  |

The code snippet starts in the global execution context. “First” is printed first. And ‘Third’ is printed from the code ‘**console.log('Third');’** as the **setTimeout(secondCall, 2000)** and **setTimeout(() => console.log('Third'), 0)** are scheduled later due to asynchronous nature and put on call stack. Net ‘Third’ printed from line and **setTimeout(() => console.log('Third'), 0).** After 2 seconds **setTimeout(secondCall, 2000)** gets executed and prints Two.

**Answer.7 JavaScript**

In JavaScript, promises are objects that represent the eventual completion (or failure) of an asynchronous operation. They are commonly used for handling asynchronous operations such as network requests or reading/writing to a database.

Promises have three different states:

1. Pending: This is the initial state of a promise. It means that the asynchronous operation is still in progress and the promise is neither fulfilled nor rejected.
2. Fulfilled: If the asynchronous operation is successful, the promise transitions to the fulfilled state. It means that the operation completed successfully, and the promise has a resulting value.
3. Rejected: If the asynchronous operation encounters an error or fails, the promise transitions to the rejected state. It means that the operation failed, and the promise contains a reason for the failure.

Here's an example of creating and using a promise in JavaScript:

**function fetchData() {**

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

**// Simulating an asynchronous operation**

**setTimeout(() => {**

**const data = { message: 'Data fetched successfully' };**

**// Resolve the promise with the data**

**resolve(data);**

**// Uncomment the following line to simulate a failure**

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

**}, 2000);**

**});**

**}**

**// Using the promise**

**fetchData()**

**.then((result) => {**

**console.log(result); // Output: { message: 'Data fetched successfully' }**

**})**

**.catch((error) => {**

**console.error(error); // Output: Error: Failed to fetch data**

**});**

In this example, the fetchData function returns a new promise. It simulates an asynchronous operation using setTimeout and resolves the promise after a delay of 2 seconds. You can uncomment the line inside the timeout to simulate a failure by rejecting the promise.

The then method is used to handle the fulfillment of the promise, and the catch method is used to handle the rejection. Depending on whether the promise is fulfilled or rejected, the appropriate callback function is executed.

**Answer.8 JavaScript**

In JavaScript, the this keyword refers to the context in which a function is executed. It allows access to the object on which a method is being invoked or the object that is currently being constructed by a constructor function. The value of this is determined at runtime and can vary depending on how a function is called.

Here's an example to illustrate the usage of the this keyword:

**// Creating an object with a method**

**const person = {**

**name: 'John',**

**greet: function() {**

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

**}**

**};**

**// Invoking the method using dot notation**

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

**// Creating a constructor function**

**function Car(make, model) {**

**this.make = make;**

**this.model = model;**

**this.start = function() {**

**console.log(`Starting ${this.make} ${this.model}.`);**

**};**

**}**

**// Creating instances of the Car object**

**const car1 = new Car('Toyota', 'Camry');**

**car1.start(); // Output: Starting Toyota Camry.**

**const car2 = new Car('Honda', 'Civic');**

**car2.start(); // Output: Starting Honda Civic.**

In the example above, the this keyword is used inside the greet method of the person object. When person.greet() is called, this refers to the person object itself, allowing access to the name property of person.

In the second part of the example, the Car constructor function is defined. Inside the constructor function, this is used to assign values to the make and model properties of the newly created object. When the start method is invoked on car1 and car2, this refers to the respective instances (car1 and car2), allowing access to their make and model properties.

It's important to note that the value of this can be influenced by how a function is invoked. For example, if a function is called without an explicit context or as a standalone function, this may refer to the global object (window in browsers, global in Node.js) or be undefined in strict mode. However, when a function is invoked as a method of an object or using the new keyword to create an instance, this is bound to the respective object or instance.

Example 2:

**// Creating an object with a method**

**const counter = {**

**count: 0,**

**increment: function() {**

**this.count++;**

**console.log(`Count: ${this.count}`);**

**}**

**};**

**// Invoking the method**

**counter.increment(); // Output: Count: 1**

**counter.increment(); // Output: Count: 2**

In this example, the counter object has a count property and an increment method. The increment method uses this to refer to the counter object itself. When counter.increment() is called, the count property of counter is incremented, and the updated count is logged to the console.

**Answer.9 JavaScript**

The event loop, call stack, callback queue, and microtask queue are key components of JavaScript's concurrency model:

1. Event Loop: The event loop is responsible for managing the execution of JavaScript code in a non-blocking manner. It continuously monitors the call stack and queues for any pending tasks, ensuring that they are executed in the appropriate order.
2. Call Stack: The call stack is a data structure that keeps track of function invocations. Whenever a function is called, a new frame is added to the top of the stack. When a function completes, its frame is removed from the stack. The call stack follows a Last-In-First-Out (LIFO) order, executing functions in a synchronous manner.
3. Callback Queue: The callback queue (also known as the task queue) is a queue that holds callbacks or tasks that are ready to be executed. These tasks usually include asynchronous callbacks, such as timers, DOM events, or AJAX responses. When a task is ready to run, it is placed in the callback queue.
4. Micro Task Queue: The microtask queue (also known as the job queue or microtask checkpoint) is a queue that holds microtasks. Microtasks are a special category of tasks that have higher priority than regular tasks in the callback queue. They are typically used for promises and other asynchronous operations. Microtasks are executed before the next task is picked from the callback queue.

**Answer.10 JavaScript**

Debouncing is a technique used in JavaScript to optimize performance by limiting the frequency of function calls, particularly in scenarios where an event is triggered rapidly and we only want to execute a function after a certain delay once the events have settled down.

When an event is triggered, the debounce function sets a timer and delays the execution of the target function. If the event is triggered again within the specified delay, the timer is reset. This process continues until the event is no longer triggered within the delay period, and then the target function is finally executed.

Here's an example project that demonstrates the use of debouncing in a search input field:

**<!DOCTYPE html>**

**<html>**

**<head>**

**<title>Debouncing Example</title>**

**<style>**

**#search-input {**

**width: 300px;**

**height: 30px;**

**font-size: 16px;**

**}**

**</style>**

**</head>**

**<body>**

**<h1>Debouncing Example</h1>**

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

**<script>**

**function debounce(func, delay) {**

**let timer;**

**return function() {**

**const context = this;**

**const args = arguments;**

**clearTimeout(timer);**

**timer = setTimeout(function() {**

**func.apply(context, args);**

**}, delay);**

**};**

**}**

**function performSearch(event) {**

**const searchText = event.target.value;**

**console.log(`Performing search for: ${searchText}`);**

**// Simulating search functionality...**

**}**

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

**const debouncedSearch = debounce(performSearch, 300);**

**searchInput.addEventListener('input', debouncedSearch);**

**</script>**

**</body>**

**</html>**

In this example, we have an input field with an ID of "search-input" and a debounce delay of 300 milliseconds. The debounce function takes the performSearch function as the target function and sets the delay. It returns a new function that will be executed when the debounced event occurs.

The performSearch function is the actual function that we want to execute after the debouncing delay. In this case, it logs the search text to the console and could potentially perform an actual search operation.

By attaching the debouncedSearch function to the "input" event listener, we ensure that the performSearch function is called only after the user has finished typing and no further input is detected within the specified delay (300 milliseconds in this case).

This debouncing technique helps to optimize performance by reducing the number of unnecessary function calls, especially in scenarios where continuous and rapid events are triggered.

**Answer.11 JavaScript**

In JavaScript, closures are an important concept that allows functions to retain access to variables from their outer lexical environment even after the outer function has finished executing. A closure is created when a function is defined inside another function and has access to its parent function's variables and scope chain.

Here's an example to illustrate closures:

**function outerFunction() {**

**const outerVariable = 'I am from the outer function';**

**function innerFunction() {**

**console.log(outerVariable);**

**}**

**return innerFunction;**

**}**

**const closure = outerFunction();**

**closure(); // Output: I am from the outer function**

In this example, the outerFunction defines an outerVariable and declares an innerFunction inside it. The innerFunction has access to the outerVariable due to the closure. When outerFunction is called and assigned to the closure variable, it returns the innerFunction. Later, when closure() is invoked, it still has access to the outerVariable and logs its value.

Following are some use cases of closures:

1. Encapsulation: Closures can be used to create private variables and functions. By defining variables and functions within a closure, they are not accessible from outside the closure, providing a way to encapsulate and protect data.
2. Data Privacy: Closures can help maintain data privacy by providing controlled access to variables. Only the functions defined within the closure can access and modify the variables, preventing direct manipulation from outside.
3. Currying and Partial Application: Closures enable currying and partial application by creating functions that remember the arguments and environment in which they were defined. This allows for creating new functions with predefined arguments, making it easier to reuse and compose functions.
4. Event Handlers: Closures are often used in event handling scenarios. By defining event handlers inside another function, closures ensure that the event handlers have access to the necessary data or variables when the event occurs.
5. Memoization: Closures can be used for caching and memoization purposes. By storing computed values within a closure, the expensive computation is performed only once, and subsequent calls to the closure return the cached value.

Closures are a powerful feature of JavaScript that enables various programming techniques and patterns. They allow for maintaining state, data privacy, and encapsulation, leading to more robust and modular code.

**Answer.12 JavaScript**

|  |  |
| --- | --- |
| github repo link | **https://github.com/GopalkrishaRao/Placement\_Assignment\_Gopalkrishna\_H\_R/tree/main/JavaScript/Q12-Blog** |
| **Deployed Pjoect link** | **https://assinment-api-js-blog-55da12.netlify.app/** |

**React**

**Answer.1 React**

React is a free and open-source front-end JavaScript library for building user interfaces based on components. Component-based archtecutre, Virtual DOM, Declarative syntax, reusability of component, unidirectional data flow etc are some of major advantages of React

**Answer.2 React**

In React, the Virtual DOM (Document Object Model) is a lightweight representation of the actual DOM. It is an abstracted copy of the real DOM that React uses to efficiently update and render UI changes.

Performance optimization, Efficient diffing, Abstracted manipulation, Platform independence, Third-party integration of libraries and frameworks are some of the advantages of Virtual DOM.

**Answer.3 React**

The lifecycle of React components refers to the series of events that occur during the creation, updating, and destruction of a component, including mounting, updating, and unmounting, which allow developers to control and manage the behavior of the component at different stages of its lifecycle.

**Answer.4 React**

Functional components in React are simpler and more lightweight, written as JavaScript functions, while class components are more feature-rich, providing additional functionality such as state management and lifecycle methods, and are written as ES6 classes.

**Answer.5 React**

Hooks are a feature in React that allow developers to use state and other React features in functional components without the need for class components. They are functions that provide a way to use state, lifecycle methods, and other React features in a more concise and reusable manner.

Hooks are specifically designed to be used in functional components. They rely on the concept of closures to maintain state between renders. Class components, on the other hand, use instance properties and lifecycle methods to manage state and other React features.

It is not possible to directly use hooks in class components, but React provides a way to use functional components alongside class components in the same application.

**Answer.6 React**

In React, lifecycle methods are predefined methods that are invoked at various stages of a component's lifecycle. These methods allow developers to control and manage the behavior of components during creation, updating, and destruction. However, it's important to note that with the introduction of React Hooks, some lifecycle methods have been deprecated in favor of using functional components with hooks.

Some of the commonly used lifecycle methods in class components are:

* componentDidMount: This method is called once, immediately after a component is mounted (inserted into the DOM tree). It is commonly used for tasks such as data fetching, setting up subscriptions, or manually modifying the DOM.
* componentDidUpdate: This method is invoked after an update to the component's state or props. It allows for performing actions based on changes in the component, such as updating the DOM or making additional data requests.
* componentWillUnmount: This method is called just before a component is unmounted and removed from the DOM. It is used for cleaning up any resources or subscriptions created in the component's lifecycle.

Advantages of using lifecycle methods in React:

* Control over component behavior: Lifecycle methods provide developers with hooks into different stages of a component's lifecycle, allowing them to control and manipulate component behavior accordingly. This helps in managing state, performing side effects, and optimizing performance.
* Proper resource management: Lifecycle methods such as componentDidMount and componentWillUnmount enable proper resource management. They allow developers to set up and clean up resources efficiently, preventing memory leaks or unnecessary subscriptions.
* Integration with external libraries: Lifecycle methods provide hooks to integrate with external libraries or perform actions that require access to the DOM
* Performance optimization: Lifecycle methods can be used to optimize performance by selectively performing actions based on changes in state or props.
* Debugging and monitoring: Lifecycle methods provide visibility into the lifecycle events of components, making it easier to debug issues or monitor component behavior during development.

**Answer.7 React**

The useState hook is a built-in hook in React that allows functional components to manage state. It provides a way to declare state variables and their setters, enabling components to maintain and update their own state.

Advantages of using the useState hook in React:

* Simplicity and readability: The useState hook simplifies state management in functional components by providing a straightforward and intuitive API. It allows developers to declare and update state variables within the component's function body, making the code more readable and concise.
* No class component requirement: Prior to hooks, managing state in React required the use of class components. With useState, functional components gain the ability to handle state, eliminating the need for class components in many cases and promoting the use of simpler and more lightweight functional components.
* Multiple state variables: The useState hook supports the declaration of multiple state variables within a single functional component. This allows for more fine-grained control over component state, leading to better organization and separation of concerns.
* Functional updates: The setter function provided by useState allows for functional updates of state variables. Instead of directly assigning a new value, you can provide a function that receives the previous state as an argument and returns the updated state. This is particularly useful when dealing with asynchronous state updates or when the new state depends on the previous state.
* Performance optimization: React's useState hook implements an efficient state update mechanism. It performs state updates using a shallow merge strategy, which means it only updates the state properties that have changed.
* Compatibility with other hooks: The useState hook is compatible with other React hooks, such as useEffect and useContext, allowing for seamless integration and composition of different hook functionalities within a single component.

**Answer.8 React**

The useEffect hook is a built-in hook in React that allows functional components to perform side effects, such as fetching data, subscribing to events, or manipulating the DOM. It is similar to lifecycle methods like componentDidMount, componentDidUpdate, and componentWillUnmount in class components.

Advantages of using the useEffect hook in React:

* Side effect management: The useEffect hook provides a convenient way to manage side effects in functional components. It allows you to specify the code that should run after each render, such as fetching data from an API or updating the DOM, without the need for class components or lifecycle methods.
* Cleaner and modular code: By encapsulating side effects within the useEffect hook, you can keep your code clean and modular. Instead of scattering side effect logic throughout your component, you can separate it into separate useEffect hooks, improving readability and maintainability.
* Dependency tracking: useEffect allows you to specify dependencies, which are the values that the effect depends on. By providing these dependencies, React will run the effect only when the dependencies have changed. This helps prevent unnecessary re-execution of side effects and ensures optimal performance.
* Support for cleanup: The useEffect hook supports a cleanup function that runs when the component is unmounted or when the effect is re-executed due to a dependency change. This allows you to clean up any resources or subscriptions that were set up during the effect's execution, preventing memory leaks or stale data.
* Integration with other hooks: useEffect can be used in conjunction with other hooks, such as useState and useContext, enabling powerful combinations of functionalities. This allows you to manage state, perform side effects, and access context within a single functional component, promoting code reuse and composition.
* Asynchronous support: useEffect supports asynchronous operations within the effect function. You can use async/await or return a Promise from the effect function to handle asynchronous tasks like data fetching in a clean and concise manner.
* Better separation of concerns: With useEffect, you can separate concerns like data fetching, event handling, and DOM manipulation into individual effects. This improves the organization of your code and makes it easier to reason about and test specific functionalities in isolation.

**Answer.9 React**

The Context API in React is a feature that allows data to be passed through the component tree without the need to pass props explicitly at every level. It provides a way to share data between components by creating a context and using the Provider and Consumer components to access and update the shared data. This simplifies the process of sharing data between components that are not directly connected in the component hierarchy and helps avoid prop drilling.

|  |  |
| --- | --- |
| github repo link |  |
| **Deployed Pjoect link** |  |

**Answer.10 React**

The useReducer hook in React is a built-in hook that allows state management using a reducer function. It is an alternative to useState and provides a way to manage more complex state logic in a predictable and organized manner.

Advantages of using useReducer in React:

* Managing complex state: useReducer is useful when state logic becomes more complex and involves multiple variables that are interdependent or have complex update logic. It allows you to handle state updates in a central reducer function, which can simplify the management of complex state transitions.
* Predictable state updates: The useReducer hook follows the principles of the Redux state management pattern. It enforces immutability and guarantees that state updates are predictable and deterministic. This makes it easier to reason about the behavior of state changes and helps prevent bugs caused by unexpected side effects.
* Better separation of concerns: By using a reducer function, useReducer enables the separation of state management from the component logic. This promotes a more modular and organized code structure, making it easier to understand and maintain the component.
* Collaboration with useContext: useReducer can be combined with the useContext hook to create a powerful state management solution. By using a context to share the dispatch function provided by useReducer, you can easily manage state across multiple components without the need for prop drilling.
* Testing: useReducer facilitates easier testing of state management logic. Since the reducer function is a pure function that takes inputs and returns outputs, it can be tested in isolation without the need for rendering the components.
* Code reusability: By encapsulating state management logic in a reducer function, you can reuse the same reducer across different components or even different parts of your application. This promotes code reusability and reduces code duplication.

It's important to note that useReducer might be overkill for simple state management scenarios where useState is sufficient. However, as the complexity of state and state transitions increases, useReducer can provide a more organized and scalable approach to state management in React applications.

**Answer.11 React**

|  |  |
| --- | --- |
| github repo link | **https://github.com/GopalkrishaRao/Placement\_Assignment\_Gopalkrishna\_H\_R/tree/main/React/Q11-%20To\_do%20app%20with%20useReducer** |
| **Deployed Pjoect link** | **https://todo-with-usereducer.vercel.app/** |

**Answer.12 React**

|  |  |
| --- | --- |
| github repo link | **https://github.com/GopalkrishaRao/Placement\_Assignment\_Gopalkrishna\_H\_R/tree/main/React/Q12-%20CounterApp** |
| **Deployed Pjoect link** | **https://react-counter-app-d432.vercel.app/** |

**Answer.13 React**

|  |  |
| --- | --- |
| github repo link | **https://github.com/GopalkrishaRao/Placement\_Assignment\_Gopalkrishna\_H\_R/tree/main/React/Q13-%20React%20Calculator** |
| **Deployed Pjoect link** | **https://react-calculator-qlwx.vercel.app/** |

**Answer.14 React**

|  |  |
| --- | --- |
| github repo link | **https://github.com/GopalkrishaRao/Placement\_Assignment\_Gopalkrishna\_H\_R/tree/main/React/Q14-%20Ract%20Tic%20Tac%20Toe%20with%20class%20component** |
| **Deployed Pjoect link** |  |

**Answer.15 React**

Prop drilling refers to the process of passing props down through multiple layers of components that do not need the props themselves, but only serve as intermediaries to pass the props to the components that actually need them. This can result in unnecessary complexity and decreased code readability.

Prop drilling can be avoided by following techniques

* Use React Context: React Context allows you to create a shared data store that can be accessed by any component within its provider. Components can directly access the data they need without the need for prop drilling. By using Context, you can provide the necessary data to components without explicitly passing the props through all the intermediary components.
* Use the useContext hook: The useContext hook is a built-in hook in React that enables components to access values from the nearest Context provider. By using useContext, you can directly access the data from the Context without the need for prop drilling.
* Component Composition: Instead of passing props down through multiple layers, you can compose components in a way that keeps the data source and the components that need the data closer together. This promotes a more direct flow of data and avoids unnecessary prop drilling.
* Lift State Up: If multiple components in a subtree need access to the same data, you can lift the state up to a common parent component. This way, the parent component can manage the state and pass it down to the child components as needed, eliminating the need for prop drilling.
* Use Redux or a state management library: Redux is a popular state management library for React applications. It provides a centralized store and allows components to access and update the state directly without prop drilling. Redux helps avoid prop drilling by providing a predictable and scalable approach to managing application state.

**Answer.16 React**

|  |  |
| --- | --- |
| github repo link | **https://github.com/GopalkrishaRao/Placement\_Assignment\_Gopalkrishna\_H\_R/tree/main/React/Q16%20TaskManager** |
| **Deployed Pjoect link** |  |

**Express**

**Answer.1 Express**

|  |  |
| --- | --- |
| github repo link |  |
| **Deployed Pjoect link** |  |

**Answer.2 Express**

In Express, middleware refers to a series of functions that are executed in the request-response cycle. Each middleware function has access to the request object (req), the response object (res), and the next middleware function in the cycle.

Middleware functions can perform various tasks, such as modifying request or response objects, logging, handling errors, authenticating requests, parsing request bodies, and more. They provide a way to extend and enhance the functionality of an Express application.

A middleware function can be defined using the app.use() or app.METHOD() functions, where METHOD is an HTTP method such as get, post, put, delete, etc.

|  |  |
| --- | --- |
| github repo link |  |
| **Deployed Pjoect link** |  |

**Answer.3 Express**

|  |  |
| --- | --- |
| github repo link |  |
| **Deployed Pjoect link** |  |

**Answer.4 Express**

Authentication is the process of verifying the identity of a user or system. It determines whether a user is who they claim to be. In other words, authentication validates the credentials provided by a user (such as username and password) against a known set of credentials stored in a system. It ensures that the user is authenticated and granted access based on their identity.

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 a user is allowed to perform or what resources they can access within a system.

**Answer.5 Express**

The difference between CommonJS and EJS modules lies in their module systems and how they handle importing and exporting modules.

1. CommonJS (CJS):

* CommonJS is a module system used in Node.js.
* It follows a synchronous approach, where modules are loaded synchronously.
* In CommonJS, modules are imported using the require function and exported using the module.exports or exports object.
* The require function is used to load modules, and it returns the exported values from the module.

1. ECMAScript Modules (ESM):
   * ECMAScript Modules is a module system introduced in ECMAScript 6 (ES6) and is supported in modern browsers and newer versions of Node.js.
   * It follows an asynchronous approach, where modules are loaded asynchronously.
   * In ESM, modules are imported using the import statement and exported using the export keyword.
   * The import statement is used to import modules, and it returns a promise that resolves to the exported values from the module.

**Answer.6 Express**

JWT stands for JSON Web Token. It is an open standard (RFC 7519) for securely transmitting information as a JSON object between parties in a compact and self-contained manner. JWTs are commonly used for authentication and authorization purposes in web applications.

A JWT consists of three parts(Header, Payload, Signature) separated by periods

With JWT, you can achieve the following:

* Stateless authentication: JWTs allow for stateless authentication. Once a user logs in and obtains a JWT, the server does not need to store session data or perform database lookups for each request. The server can verify the authenticity and integrity of the token using the secret key or public key, allowing the user to access protected resources.
* Single sign-on (SSO): JWTs can be used for implementing single sign-on across multiple applications. Once a user is authenticated in one application and obtains a JWT, they can use the same token to access other applications in the ecosystem without the need for separate login credentials.
* Authorization and access control: JWTs can carry information about user roles, permissions, or access levels. This allows the server to easily determine what actions or resources a user is authorized to access. It simplifies the process of implementing fine-grained access control and authorization checks.
* Information exchange: JWTs can be used to securely exchange information between different systems or services. The claims in the payload can contain data that needs to be shared between parties, such as user information or custom data. The information in the token is signed, ensuring its integrity and preventing tampering.
* Cross-domain communication: JWTs can be used to facilitate communication between different domains or services in a secure manner. By including the JWT in the HTTP headers or as a parameter in API requests, the receiving service can validate and trust the token to establish trust and ensure secure communication.
* Revocation and expiration: JWTs can be issued with an expiration time, after which they are no longer considered valid. This allows for automatic session expiration and reduces the need for server-side storage or session management. In case of revocation or logouts, JWTs can be added to a blacklist or invalidated by changing the server-side secret key.

|  |  |
| --- | --- |
| github repo link |  |
| **Deployed Pjoect link** |  |

**Answer.7 Express**

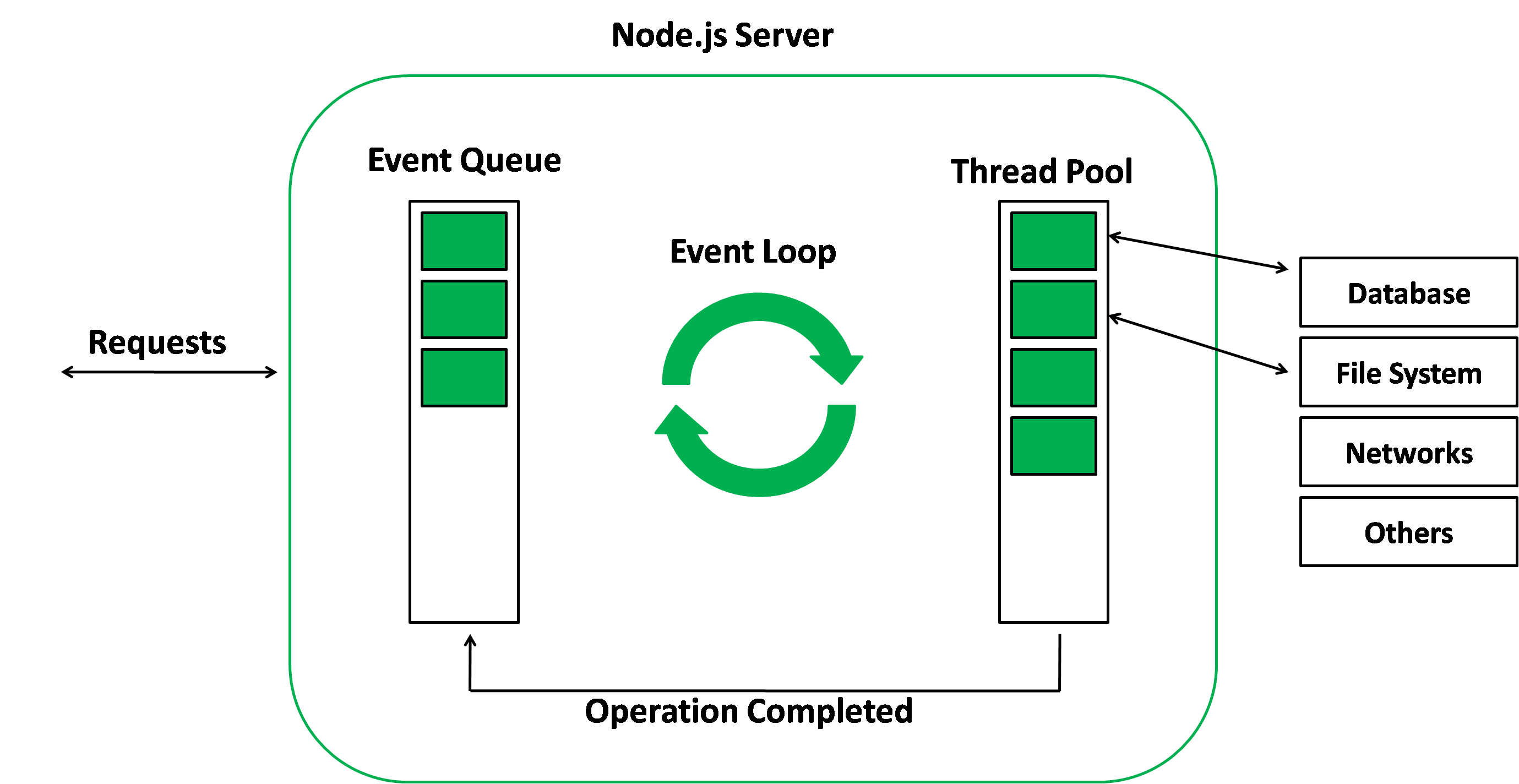
When storing a user's password in a database, it is essential to follow secure practices to protect the password from unauthorized access. Here are recommended steps to handle passwords securely:

* Hashing: Hashing is a one-way mathematical function that converts a password into a fixed-length string of characters. Use a strong and slow hashing algorithm such as bcrypt, Argon2, or scrypt. These algorithms are designed to be computationally expensive, making it difficult for attackers to guess the original password from the hash.
* Salting: Salting is the process of adding a unique random string (called a salt) to the password before hashing. The salt adds extra randomness to each password, making it harder to crack through precomputed hash tables (rainbow tables) or brute force attacks. The salt should be unique per user and securely stored alongside the hashed password.

**Answer.8 Express**

The event loop is a fundamental concept in Node.js that enables asynchronous, non-blocking I/O operations. It is responsible for handling and executing events and callbacks in an event-driven, single-threaded environment.

In Node.js, the event loop allows the server to handle multiple concurrent requests efficiently without blocking the execution of other operations. It ensures that I/O operations, such as reading from or writing to files, making network requests, or querying databases, can be performed asynchronously without blocking the execution of other code.



Event loop works in Node.js as follows:

* Event Loop Initialization: When Node.js starts, it initializes the event loop.
* Event Registration: Asynchronous operations, such as reading a file or making an HTTP request, are initiated by registering event handlers or callbacks. These callbacks are associated with specific events, such as data being available or an operation completing.
* Event Queue: When an asynchronous operation completes or an event occurs, the associated callback is placed in the event queue.
* Event Loop Phases: The event loop continuously iterates over a set of predefined phases, checking for any pending events or callbacks in the event queue. Each phase has a specific set of callbacks it handles.
* Event Processing: In each iteration of the event loop, it checks the event queue for pending callbacks. If there are callbacks in the queue, they are executed one by one in the order they were registered.
* Blocking Operations: If a synchronous (blocking) operation is encountered during the event loop, it temporarily pauses the execution of the event loop until the blocking operation completes. This can disrupt the event loop's responsiveness and should generally be avoided.
* Timers: The event loop also handles timers, such as setTimeout and setInterval. It schedules the execution of timer callbacks based on their specified delay.

By utilizing the event loop, Node.js can efficiently handle multiple concurrent operations and scale well under high loads. It maximizes CPU utilization by minimizing idle time and ensures that applications remain responsive even during I/O-intensive tasks.

**Answer.9 Express**

|  |  |
| --- | --- |
| github repo link |  |
| **Deployed Pjoect link** |  |