PLACEMENT ASSISTANCE PROGRAM INTERVIEW BASED ASSIGNMENT

HTML Questions

Question 1

<!DOCTYPE html> Is it a tag of html? If not, what is it and why do we use it?

Ans:

<!DOCTYPE html> is not a HTML tag. It is a declaration made to inform the browser about the HTML version used in the document. If the user application experiences any problems it can be quickly determined that the problem is due to incompatibility between the HTML version and the user's browser. It is known as Document Type Declaration (DTD).

Question 2

Explain Semantic tags in html? And why do we need it?

Ans:

Semantic tags in HTML refer to the use of specific HTML elements that convey meaning and structure to the content of a web page. These tags are designed to describe the purpose or role of the content they wrap, rather than just determining how it should be presented visually. Semantic tags help search engines, assistive technologies, and other web tools understand the content better and provide a more meaningful experience to users. Some examples of semantic tags are :- <header>, <nav>, <article>, <footer>, <aside> etc.

Question 3
Differentiate between HTML Tags and Elements?

Ans:

1. HTML Tags: HTML tags are the markup characters used to define elements in an HTML document. They are enclosed in angle brackets (< and >) and appear as pairs, consisting of an opening tag and a closing tag. For example, <p> is the opening tag, and </p> is the closing tag for a paragraph element.
2. HTML Elements: HTML elements are made up of HTML tags along with the content they wrap. An HTML element consists of an opening tag, the content, and a closing tag (if applicable).

Question 4
Build Your Resume using HTML only

Ans:

Github Repo Link :
https://github.com/Pritika17/Placement-Assignment-Pritika-Mishra-htmlQ4

Question 5
Write Html code so that it looks like the given image Link

Ans:

Github Repo Link :

https://github.com/Pritika17/Placement-Assignment-PritikaMishra-html

Question 6

What are some of the advantages of HsTML5 over its previous versions?

Ans:

1. Improved semantics with new tags like <header> and <nav>.
2. Native support for multimedia elements (video, audio) without plugins.
3. Mobile-friendly features like responsive design and input types.
4. Offline support with local data storage (Web Storage API).
5. Enhanced form handling and validation.
6. Simplified markup, leading to cleaner code.
7. Accessibility improvements with aria-* attributes.
8. Performance enhancements with new APIs and asynchronous loading.

Question 7

Create a simple Music player using html only

Ans:

GitHub Repo Link:
https://github.com/Pritika17/Placement-Assignment-PritikaMishra-htmlQ7

Question 8

What is the difference between <figure> tag and <img> tag?

Ans:

The <img> tag is an HTML element used to display standalone images on a web page. It requires the src attribute to specify the image source. On the other hand, the <figure> tag is used to group together a self-contained piece of content, such as an image, along with an optional caption defined using the <figcaption> tag.
The <figure> element provides a semantic way to associate descriptive text with related media.

Question 9

What's the difference between html tag and attribute and give examples of some global attributes?

Ans:

HTML Tag:

An HTML tag is the markup element used to define the structure and content of an HTML document. Tags are enclosed in angle brackets (< and >) and can have attributes that provide additional information about the tag or modify its behavior. Examples of HTML tags include <div>, <p>, <h1>, and <img>.

Attribute:

An attribute is a property or characteristic of an HTML element that provides additional information or modifies the element's behavior. Attributes are specified within the opening tag of an HTML element and consist of a name-value pair. Examples of attributes include class, id, src, and href.

Global Attributes:

Global attributes are attributes that can be used with any HTML element, irrespective of the specific tag. Here are some examples of global attributes:

- class: Specifies CSS classes for styling.
- id: Provides a unique identifier for an element.
- style: Defines inline CSS styles.

- title: Displays additional information or tooltip text.
- data-*: Stores custom data for JavaScript or CSS use.
- aria-*: Used for accessibility purposes.

Question 10

build Table which looks like the given image

GitHub Repo Link:
https://github.com/Pritika17/Placement-Assignment-PritikaMishra-htmlQ10

CSS Questions

Question 1

What's Box Model in CSS & Which CSS Properties are part of it ?

Ans:

The box model in CSS defines the layout and spacing properties of an element. It consists of four components: content, padding, border, and margin.

CSS properties that are part of the box model:

- width and height define the dimensions of the content box.
- padding adds space between the content and the border.
- border sets the style, width, and color of the border around the content and padding.
- margin creates space outside the border, separating the element from other elements.

Question 2

What are the Different Types of Selectors in CSS & what are the advantages of them?

Ans:

Types of Selectors in CSS

1. Element selectors: They target HTML elements based on their tag name. For example, p selects all <p> elements.
2. Class selectors: They target elements with a specific class attribute. For example, .my-class selects all elements with the class "my-class".
3. ID selectors: They target elements based on their unique ID attribute. For example, #my-id selects the element with the ID "my-id".
4. Attribute selectors: They target elements based on specific attribute criteria. For example, [type="text"] selects elements with the attribute type set to "text".
5. Descendant selectors: They target elements that are descendants of another element. For example, div p selects all <p> elements that are descendants of <div> elements.
6. Pseudo-classes and pseudo-elements: Pseudo-classes target elements based on states or occurrences, such as :hover for hovering over an element. Pseudo-elements target specific parts of an element, like ::first-line for styling the first line of text.

Advantages are as follows:

- Flexibility: Different selectors allow you to target specific elements or groups of elements based on various criteria, providing flexibility in styling and customization.
- Specificity: Selectors like class and ID selectors offer higher specificity, allowing you to override general styles or target specific elements with precision.
- Reusability: Class selectors enable you to define styles once and apply them to multiple elements, promoti`ng code reusability and consistency.
- Maintenance: Selectors help organize your CSS code and make it easier to understand and maintain, especially when used with clear naming conventions.
- Interaction: Pseudo-classes and pseudo-elements enable you to add interactivity and dynamic effects to your elements, enhancing the user experience.

Question 3

What is VW/VH & How is it different from PX?

Ans:

- VW (Viewport Width) and VH (Viewport Height) are relative units of measurement in CSS.
- They are based on the dimensions of the viewport, which is the visible area of a web page.
- VW represents a percentage of the viewport's width, while VH represents a percentage of the viewport's height.
- Advantages of using VW and VH units include responsiveness and
- scalability. Elements sized with VW and VH units automatically adjust in size based on the viewport dimensions.
- This makes VW and VH ideal for creating responsive layouts that adapt to different screen sizes and resolutions.
- On the other hand, PX (pixels) represent an absolute unit of measurement providing fixed sizes.
- PX units do not respond to changes in viewport size.
- This can result in elements that do not scale well on different devices or when the viewport size changes.
- VW and VH units offer a flexible and adaptable approach to element sizing, while PX units provide precise control over fixed sizes.

Question 4

What's the difference between Inline, Inline Block and block ?

Ans:

The main differences between the display properties of inline, inline-block, and block elements are as follows:

Inline Elements:

- Inline elements do not start on a new line. They flow within the text content and do not create line breaks.
- They only take up the necessary width and height to accommodate their content.
- Examples of inline elements include <span>, <a>, <strong>, and <em>.

Inline-Block Elements:

- Inline-block elements behave like inline elements in terms of flowing within the text content.
- However, they can have a specified width, height, padding, and margin properties like block elements.
- Inline-block elements allow for the coexistence of multiple elements on the same line while still respecting their specified dimensions.
- Examples of inline-block elements include <img>, <input>, and <button>.

Block Elements:

- Block elements start on a new line and create line breaks before and after themselves.
- They take up the full available width by default and can have a specified width and height.
- Block elements can have margin, padding, and border properties, allowing for more complex layouts.
- Examples of block elements include <div>, <p>, <h1> to <h6>, and <ul>.

Question 5

How is Border-box different from Content Box?

Ans:

The main difference between border-box and content-box is how they calculate the total width and height of an element, including its content, padding, and border.

1. **Content-box:**
- By default, elements use the content-box box-sizing property.
- The total width and height of an element are calculated by adding the content width and height, as well as any padding and border, which increases the overall size of the element.
- In other words, the width and height specified for the element represent the content area only, and the padding and border are added to the specified width and height.
2. **Border-box:**
- When using the border-box box-sizing property, the total width and height of an element are calculated differently.

- In this case, the specified width and height of the element include both the content area, padding, and border.
- The browser automatically adjusts the content width and height to accommodate the padding and border, ensuring that the specified width and height remain unchanged.
- This means that the padding and border are included within the specified width and height, and they do not increase the overall size of the element.

Question 6

What’s z-index and How does it Function ?

Ans:

z-index is a CSS property that controls the stacking order of elements on a web page along the z-axis, which represents depth or elevation in the 3D space.

The z-index property assigns a numerical value to an element, determining its position in the stacking order relative to other elements. The higher the z-index value, the closer the element is to the viewer, and the more it will appear on top of other elements.

The z-index property can only be applied to positioned elements (those with a position value of relative, absolute, or fixed). By default, all elements have a z-index value of auto, which means they stack in the order they appear in the HTML structure.

Here's how z-index functions:

1. Elements with a higher z-index value will be placed in front of elements with a lower value.
2. If two elements have the same z-index, the stacking order will follow the order in the HTML structure (the element appearing later will be on top).
3. If an element has a negative z-index, it will be positioned behind elements with positive or auto z-index values.

Question 7

What’s Grid & Flex and difference between them?

Ans:

Grid and Flex are both CSS layout systems used to create responsive and flexible designs, but they have different approaches and use cases.

Grid:

- CSS Grid is a two-dimensional layout system that allows for the creation of complex grid-based layouts.
- It divides a web page into columns and rows, allowing precise control over the placement and alignment of elements within the grid.
- Grid provides powerful features like grid lines, grid tracks, grid areas, and grid templates, allowing for flexible and responsive designs.
- It is well-suited for creating complex layouts with multiple rows and columns, such as magazine-style websites, grid-based galleries, and overall page structure.

Flexbox:

- CSS Flexbox is a one-dimensional layout system that focuses on arranging elements along a single axis (either horizontally or vertically).
- It is designed for simpler layouts and aims to provide flexibility in distributing space and aligning elements within a container.
- Flexbox uses the concept of flex containers and flex items, where the container controls the layout and positioning of the items.
- Flexbox offers features like flexible widths, alignments, and space distribution, making it ideal for creating responsive navigation menus, equal-height columns, and vertically aligning elements.

Key Differences:

1. Dimension: Grid is a two-dimensional layout system, while Flexbox is a one-dimensional layout system.
2. Layout Control: Grid provides fine-grained control over both columns and rows, allowing for complex layouts. Flexbox focuses on arranging items along a single axis within a container.
3. Use Cases: Grid is suitable for creating intricate and structured layouts with multiple rows and columns. Flexbox is great for simpler layouts and aligning items along a single axis.

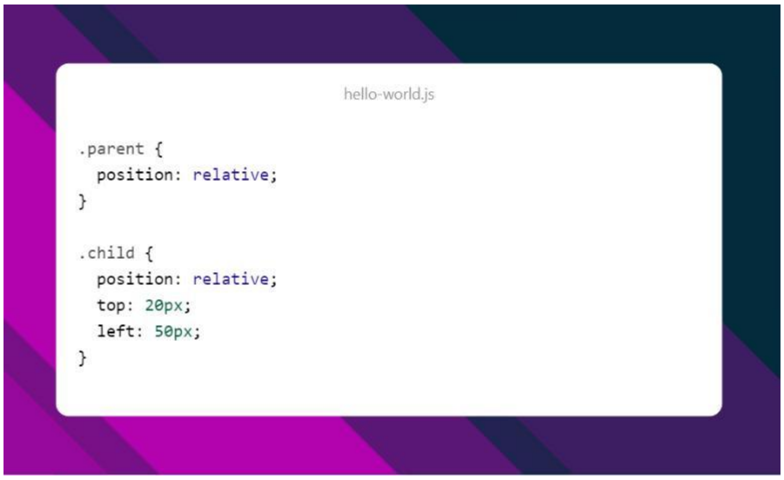
Question 8

Difference between absolute and relative and sticky and fixed position explain with example.

Ans:

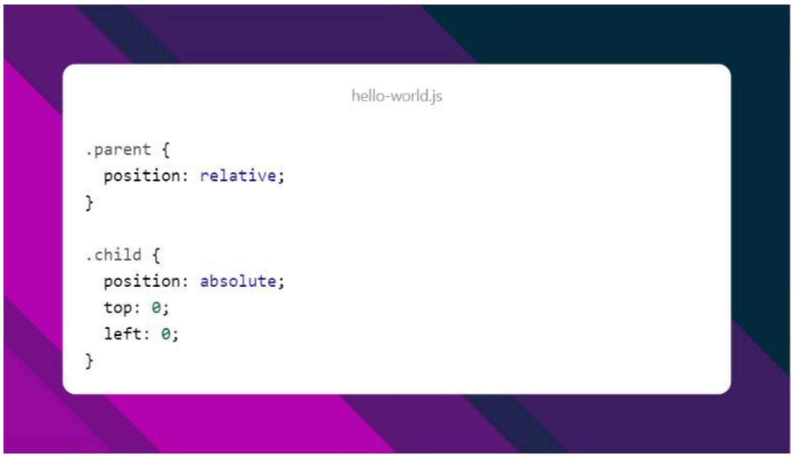
Absolute and Relative Positions:

- Relative position: When an element has a relative position, it is positioned relative to its normal position within the document flow. It can be moved using properties like top, bottom, left, and right. Other elements will still respect the space taken by the relatively positioned element.0 Example:



```
.parent {
  position: relative;
}

.child {
  position: relative;
  top: 20px;
  left: 50px;
}
```

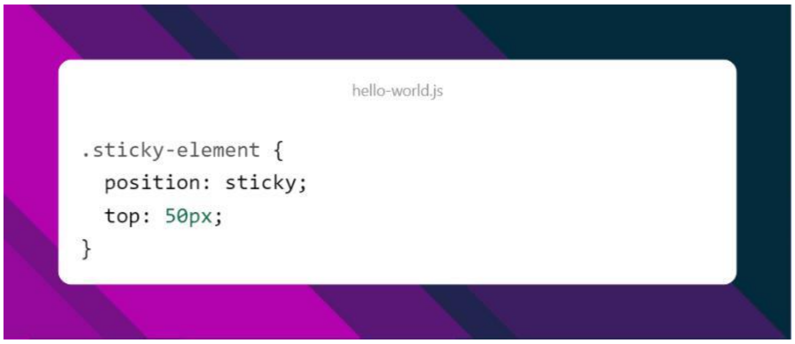
- Absolute position: When an element has an absolute position, it is positioned relative to its nearest positioned ancestor (if any), or the initial containing block (if no positioned ancestor exists). It is taken out of the normal document flow, and other elements will ignore the space it occupies.0 Example:



```
.parent {
  position: relative;
}

.child {
  position: absolute;
  top: 0;
  left: 0;
}
```

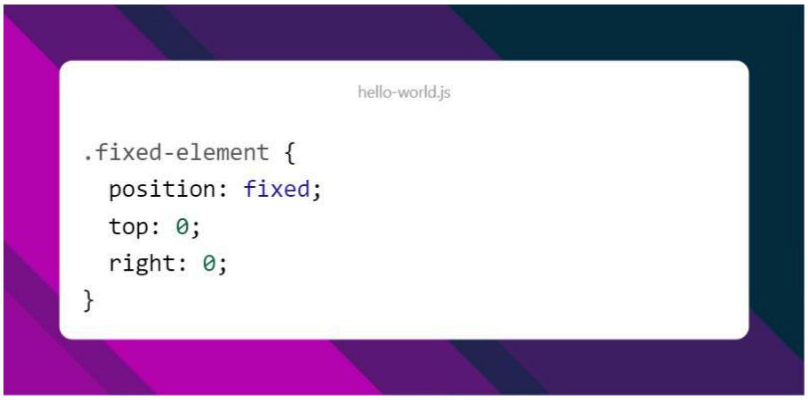
Sticky and Fixed Positions:

- Sticky position: When an element has a sticky position, it is initially positioned according to the normal flow of the document. However, as the user scrolls, it "sticks" to a specified position on the screen. It toggles between being positioned relative and fixed, depending on the scroll position.0 Example:



```
.sticky-element {
  position: sticky;
  top: 50px;
}
```

- Fixed position: When an element has a fixed position, it is positioned relative to the viewport and remains fixed at a specified position, even when the page is scrolled. It does not move when the user scrolls the page.0 Example:



```
.fixed-element {
  position: fixed;
  top: 0;
  right: 0;
}
```

Question 9
Build Periodic Table as shown in the below image

GitHub Repo Link:
https://github.com/Pritika17/Placement-Assignment-PritikaMishra-CSSQ8

Question 10
Build given layout using grid or flex see below image for reference .

GitHub Repo Link:
https://github.com/Pritika17/Placement-Assignment-PritikaMishra-CSSQ10

Question 11
Build Responsive Layout both desktop and mobile and Tablet, see below image for reference ?

GitHub Repo Link:
https://github.com/Pritika17/Placement-Assignment-PritikaMishra-CSSQ11

Question 12
What are Pseudo class in CSS & How its different From Pseudo Elements?

Ans:

In CSS, pseudo-classes and pseudo-elements are used to select and style specific elements based on certain conditions or positions. While they have similar names, they serve different purposes.

Pseudo-classes:

A pseudo-class is used to select elements based on their state or interaction. It targets elements that are in a specific state or fulfill a particular condition. Pseudo-classes are preceded by a colon (":") in CSS. Here are a few examples of common pseudo-classes:

1. :hover - Selects an element when the mouse cursor is over it.
2. :active - Selects an element while it is being activated (such as clicking a button).
3. :focus - Selects an element when it is receiving focus (such as when an input field is selected).
4. :first-child - Selects the first child element of its parent.
5. :nth-child - Selects elements based on their position in relation to their parents or siblings.

Pseudo-elements:

A pseudo-element is used to style a specific part of an element's content or structure. It allows you to create virtual elements that don't exist in the HTML markup but can be targeted and styled using CSS. Pseudo-elements are preceded by a double colon ("::") in CSS. Here are a few examples of common pseudo-elements:

1. ::before - Inserts content before the selected element.
2. ::after - Inserts content after the selected element.
3. ::first-line - Selects the first line of text within an element.
4. ::first-letter - Selects the first letter of text within an element.
5. ::selection - Selects the portion of text that has been highlighted by the user.

The main difference between pseudo-classes and pseudo-elements is that pseudo-classes select and style whole elements based on their state, while pseudo-elements target specific parts of an element's content or structure.

Pseudo-classes are denoted by a single colon (":") and pseudo-elements by a double colon ("::"). However, due to historical reasons, some pseudo-elements (such as ::before and ::after) can also be written with a single colon, and both forms are usually considered valid.

JavaScript Questions

Question 1

What is Hoisting in Javascript ?

Ans:

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 they are placed in the code, they are effectively "hoisted" to the top. However, it's important to note that only the declarations are hoisted, not the assignments or initializations. This allows variables to be used before they are declared and functions to be called before they are defined. To ensure code clarity and avoid potential issues, it is considered a best practice to declare variables and functions at the top of their respective scopes, even though hoisting may allow them to be used elsewhere in the code.

Question 2

What are different higher order functions in JS? What is the difference between .map() and .forEach() ?

Ans:

In JavaScript, higher-order functions are functions that take other functions as arguments or return functions as results. They provide a way to work with functions as data, enabling powerful functional programming patterns. Some common higher-order functions in JavaScript include map(), forEach(), filter(), reduce(), and sort(), among others.

Now, let's focus on the difference between map() and forEach():

1. **map():**0
   The map() method creates a new array by calling a provided function on each element of the original array. It applies the provided function to every element

and returns a new array with the results in the same order. The original array remains unchanged. The map() function is useful when you want to transform each element of an array into a new value.

2. **forEach()**:0
   The forEach() method iterates over the elements of an array and executes a provided function for each element. Unlike map(), it does not create a new array or return any value. It is typically used when you want to perform some operation or side effect on each element of an array, without creating a new array.

Key differences:

- Return value: map() returns a new array with the results of the function applied to each element, while forEach() does not return anything. Side
- effects: forEach() is commonly used for performing actions or side effects on each element, such as console logging or modifying external variables. map(), on the other hand, is purely used for transforming elements into a new value.
- Immutability: map() creates a new array without modifying the original one, while forEach() does not create a new array or modify the original array.
- Chaining: Since map() returns a new array, it can be easily chained with other array methods. forEach() does not produce a return value, so chaining it with other methods is not typical.

Question 3

What is the difference between .call() .apply() and .bind()? explain with an example

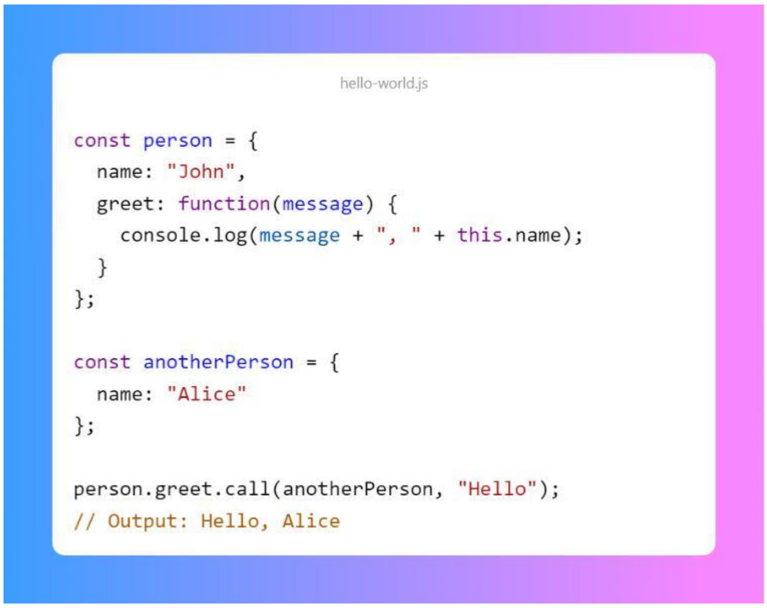
Ans:

In JavaScript, the methods .call(), .apply(), and .bind() are used to manipulate the value of this keyword in a function and to invoke a function in a specific context. While they serve a similar purpose, there are subtle differences between them.

1. .call():0
   The .call() method allows you to invoke a function with a specified this value and individual arguments passed directly. It takes the this value as the first

parameter, followed by any additional arguments as comma-separated values.

Example:



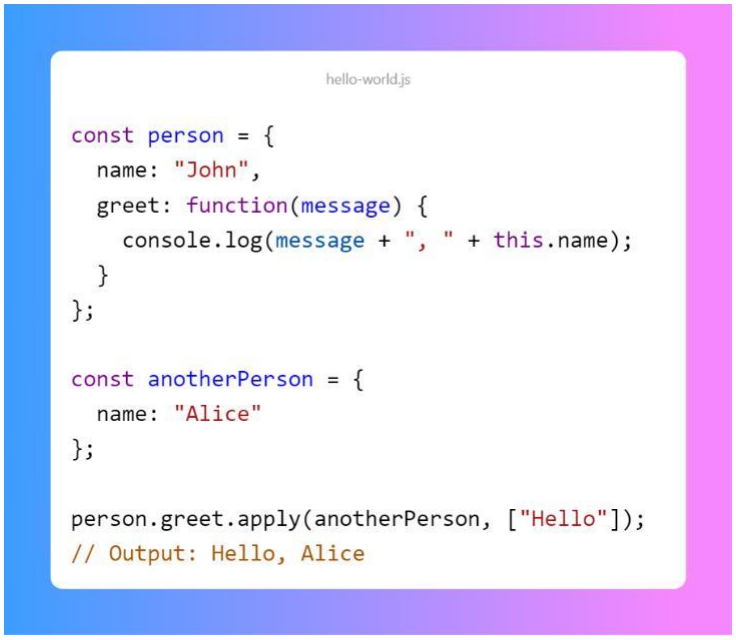
hello-world.js

```javascript
const person = {
  name: "John",
  greet: function(message) {
    console.log(message + ", " + this.name);
  }
};

const anotherPerson = {
  name: "Alice"
};

person.greet.call(anotherPerson, "Hello");
// Output: Hello, Alice
```

In the example above, .call() is used to invoke the greet() method of the person object with anotherPerson as the this value.

1. .apply():0
   The .apply() method is similar to .call(), but it accepts arguments as an array or an array-like object instead of individual values. It also allows you to specify the this value for the function.

Example:



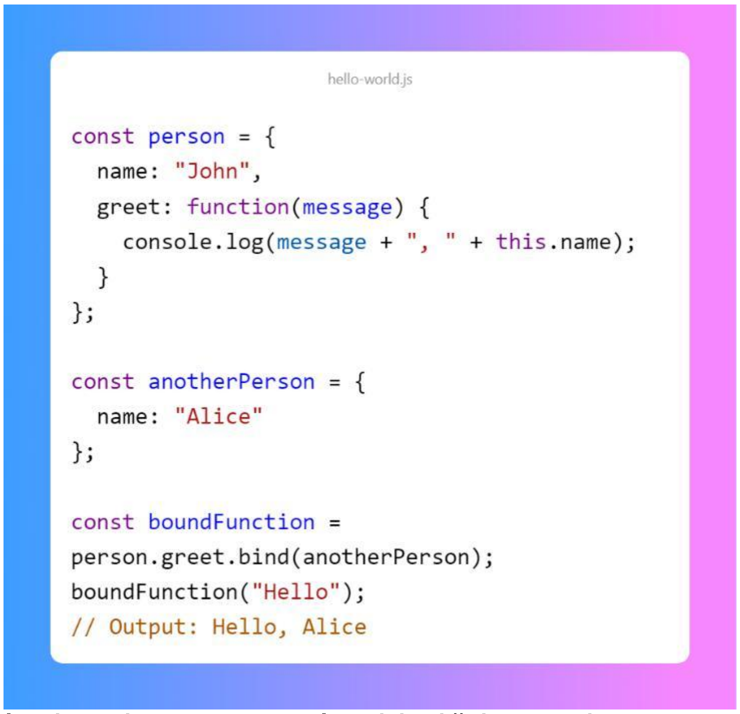
hello-world.js

```javascript
const person = {
  name: "John",
  greet: function(message) {
    console.log(message + ", " + this.name);
  }
};

const anotherPerson = {
  name: "Alice"
};

person.greet.apply(anotherPerson, ["Hello"]);
// Output: Hello, Alice
```

In this example, .apply() is used to invoke the greet() method with anotherPerson as the value and the message passed as an array.

1. .bind():0
   The .bind() method creates a new function with a specified this value and any initial arguments. It returns a new function that, when invoked, has the specified context and the provided arguments. Unlike .call() and .apply(), .bind() doesn't immediately invoke the function.

Example:



```javascript
const person = {
  name: "John",
  greet: function(message) {
    console.log(message + ", " + this.name);
  }
};

const anotherPerson = {
  name: "Alice"
};

const boundFunction =
person.greet.bind(anotherPerson);
boundFunction("Hello");
// Output: Hello, Alice
```

In the above example, .bind() is used to create a new function boundFunction with the this value set to anotherPerson. When boundFunction is invoked, it logs the message with anotherPerson's name.

Key differences:

- .call() and .apply() immediately invoke the function, while .bind() returns a new function.
- .call() accepts individual arguments, .apply() accepts an array of arguments, and .bind() accepts arguments individually and returns a new function.
- .call() and .apply() can only be used once, while .bind() creates a new function that can be invoked multiple times.

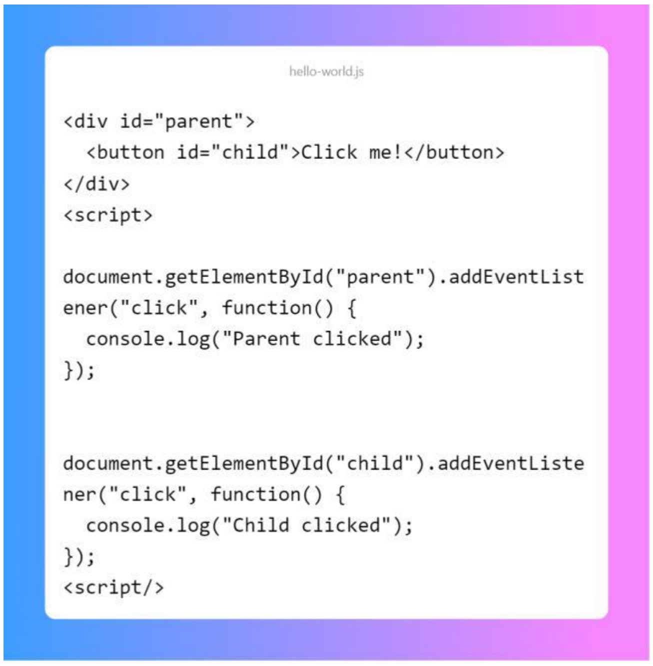
Question 4
Explain Event bubbling and Event Capturing in JavaScript with suitable examples

Ans:

Event bubbling and event capturing are two different mechanisms that describe how events propagate through the DOM (Document Object Model) hierarchy in JavaScript.

1. Event Bubbling:0
   Event bubbling is the default behavior in which an event is first handled at the target element, and then it propagates up the DOM tree to its ancestors. This means that when an event occurs on an element, such as a button click, the event handler for that element is called first, followed by the event handlers of its parent elements.

Example:



hello-world.js

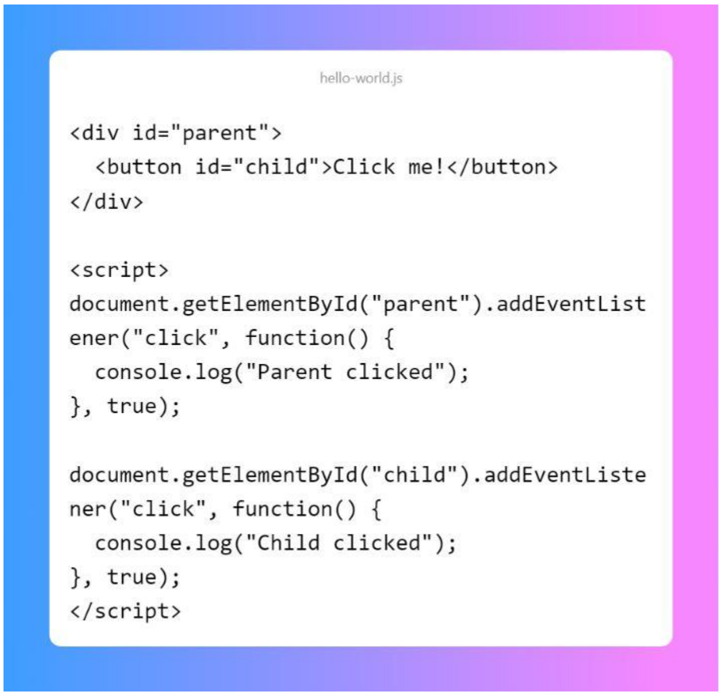
```
<div id="parent">
  <button id="child">Click me!</button>
</div>
<script>

document.getElementById("parent").addEventList
ener("click", function() {
  console.log("Parent clicked");
});


document.getElementById("child").addEventListe
ner("click", function() {
  console.log("Child clicked");
});
<script/>
```

When the "Click me!" button is clicked, both event handlers will be executed. First, "Child clicked" will be logged, followed by "Parent clicked". This is because the event bubbles up from the child element to its parent.

1. Event Capturing:0
   Event capturing is the opposite mechanism, where the event is first captured at the top of the DOM tree and then propagates down to the target element.

In event capturing, the event handlers of the parent elements are triggered before the event reaches the target element.

Example:



hello-world.js

```
<div id="parent">
  <button id="child">Click me!</button>
</div>

<script>
document.getElementById("parent").addEventList
ener("click", function() {
  console.log("Parent clicked");
}, true);

document.getElementById("child").addEventListe
ner("click", function() {
  console.log("Child clicked");
}, true);
</script>
```

With event capturing enabled by passing true as the third parameter of addEventListener, when the "Click me!" button is clicked, the event handlers will be executed in reverse order. First, "Parent clicked" will be logged, followed by "Child clicked".

By default, event handling in JavaScript uses event bubbling. However, you can explicitly enable event capturing by setting the third parameter of addEventListener to true

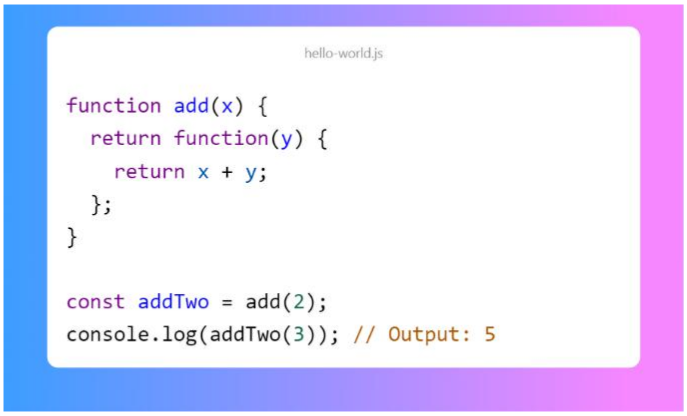
Question 5

What is the function currying with an example?

Ans:

Function currying is a technique in JavaScript where a function with multiple arguments is transformed into a series of functions, each taking a single argument. This allows for partial application, where you can provide some arguments upfront and obtain a new function that expects the remaining arguments. It enables more flexible and reusable code.

Here's an example:



```
function add(x) {
  return function(y) {
    return x + y;
  };
}

const addTwo = add(2);
console.log(addTwo(3)); // Output: 5
```

In this example, the add() function takes an argument x and returns an inner function that takes another argument y. By partially applying add(2), we obtain a new function addTwo that expects the y parameter. When we invoke addTwo(3), it adds 2 (from the partial application) and 3, resulting in the output 5.

Question 6

Explain execution context diagram of following code snippets, use white board to draw execution context diagram

Code Snippet 1

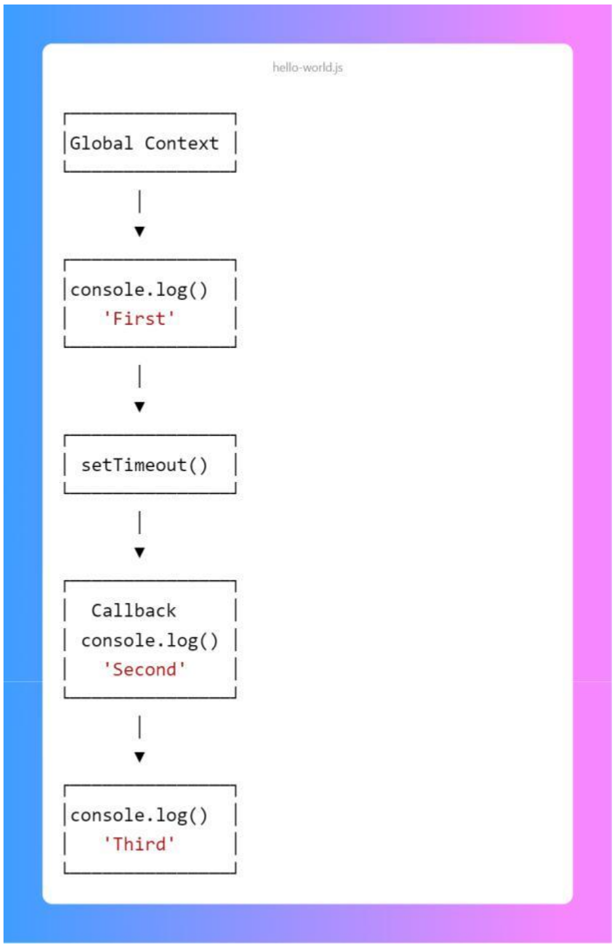
```
console.log('First');
setTimeout(() => console.log('Second'), 0);
console.log('Third');
```

Code Snippet 2

```
console.log('First');
function secondCall() {
console.log('Second');
}
setTimeout(secondCall, 2000);
setTimeout(() => console.log('Third'), 0);
console.log('Third');
```

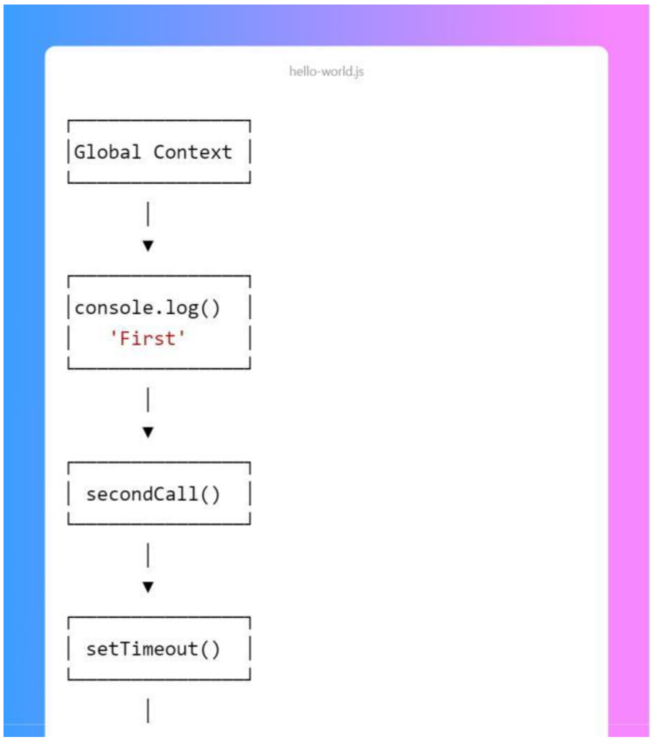
Ans:

Code snippet 1:

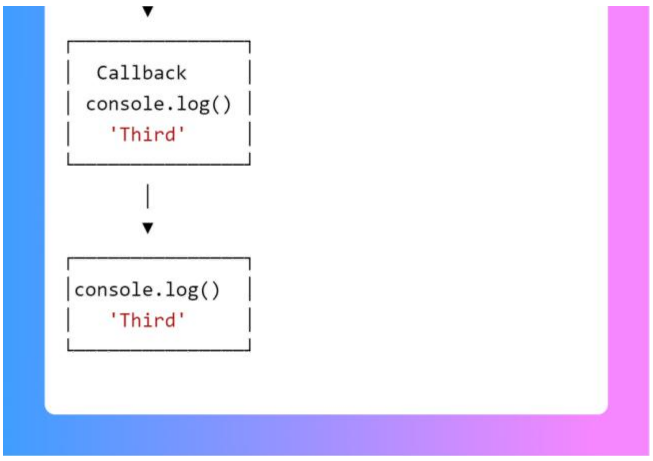


```
hello-world.js

┌──────────────┐
│Global Context │
└──────────────┘
      │
      ▼
┌──────────────┐
│console.log()  │
│  'First'      │
└──────────────┘
      │
      ▼
┌──────────────┐
│ setTimeout()  │
└──────────────┘
      │
      ▼
┌──────────────┐
│  Callback     │
│ console.log() │
│  'Second'     │
└──────────────┘
      │
      ▼
┌──────────────┐
│console.log()  │
│  'Third'      │
└──────────────┘
```

Code Snippet 2:



```
hello-world.js

┌──────────────┐
│Global Context │
└──────────────┘
      │
      ▼
┌──────────────┐
│console.log()  │
│  'First'      │
└──────────────┘
      │
      ▼
┌──────────────┐
│ secondCall()  │
└──────────────┘
      │
      ▼
┌──────────────┐
│ setTimeout()  │
└──────────────┘
      │
```



These diagrams follow a similar style to the ones used in the MDN documentation, illustrating the sequence of execution from the global context to the subsequent function calls and console.log statements. The arrows represent the flow of control as the execution progresses through each step.

Question 7

What are promises? What are the different states of a promise? Support your answer with an example where you need to create your own promise.

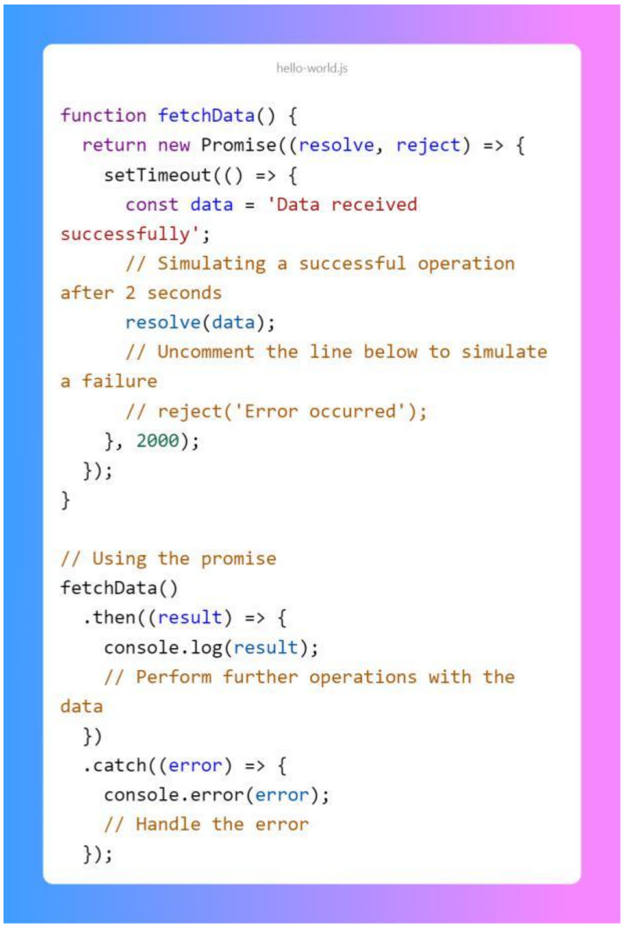
Ans:

Promises are a way to handle asynchronous operations in JavaScript. They represent the eventual completion or failure of an asynchronous operation and allow you to write asynchronous code in a more readable and manageable way.

Promises have three different states:

1. **Pending**: The initial state of a promise. It means that the asynchronous operation associated with the promise is still ongoing and has not been fulfilled or rejected yet.
2. **Fulfilled**: The state of a promise when the asynchronous operation has completed successfully. It means that the promised value is available and can be used in further operations.
3. **Rejected**: The state of a promise when the asynchronous operation encounters an error or fails. It means that the promised value is not available due to an error or failure.

Here's an example where you create your own promise to simulate an asynchronous operation:



```
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const data = 'Data received
successfully';
      // Simulating a successful operation
after 2 seconds
      resolve(data);
      // Uncomment the line below to simulate
a failure
      // reject('Error occurred');
    }, 2000);
  });
}

// Using the promise
fetchData()
  .then((result) => {
    console.log(result);
    // Perform further operations with the
data
  })
  .catch((error) => {
    console.error(error);
    // Handle the error
  });
```

In this example, the fetchData function returns a new Promise. Inside the promise constructor, there's a setTimeout function that simulates an asynchronous operation. After 2 seconds, the promise either resolves with the data or rejects with an error.

Using the promise, we can chain the then method to handle the fulfilled state and the catch method to handle the rejected state. If the promise resolves successfully, we receive the data in the then callback. If it rejects, we catch the error in the catch callback.

This example demonstrates how promises provide a structured way to handle asynchronous operations and handle both successful and error scenarios.

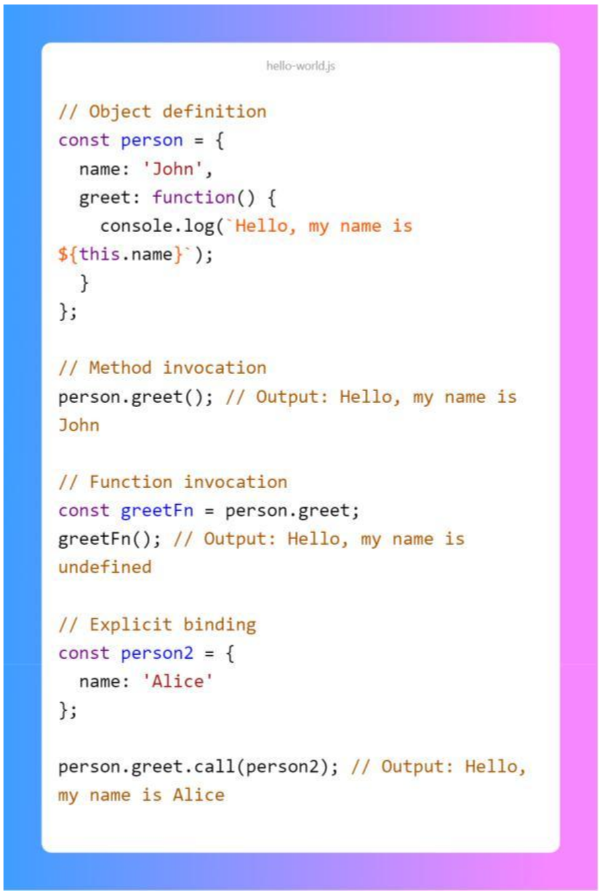
Question 8

What is ‘this’ keyword in JavaScript? explain with an example & create

Ans:

In JavaScript, the this keyword refers to the context in which a function is invoked. It allows access to the object that owns or calls the function. The value of this is determined dynamically at runtime and can vary depending on how a function is called.

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



```
// Object definition
const person = {
  name: 'John',
  greet: function() {
    console.log(`Hello, my name is
${this.name}`);
  }
};

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

// Function invocation
const greetFn = person.greet;
greetFn(); // Output: Hello, my name is
undefined

// Explicit binding
const person2 = {
  name: 'Alice'
};

person.greet.call(person2); // Output: Hello,
my name is Alice
```

In the example above, we have an object person with a name property and a greet method. When the greet method is invoked as a method of the person object (person.greet()), this inside the method refers to the person object itself. Therefore, this.name accesses the name property of the person object.

However, when we assign the greet method to the greetFn variable and invoke it as a standalone function (greetFn()), this inside the function does not reference the person object. Instead, it becomes the global object (window object in a browser or global object in Node.js). In this case, this.name results in undefined because the global object does not have a name property.

To explicitly bind this to a specific object, we can use the call() method or the apply() method. In the example, we create a new object person2 and use the call() method to invoke the greet method with this set to person2. This way, this.name now accesses the name property of person2.

Question 9

Explain event loop Call Stack Callback queue and Micro Task queue in Your Words

Ans:

1. **Call Stack**: The call stack is a data structure that keeps track of function calls in JavaScript. When a function is called, it is added to the top of the call stack. The function execution begins, and when it finishes, it is removed from the call stack. This stack-like behavior allows JavaScript to keep track of the execution order of functions.
2. **Event Loop**: The event loop is a crucial component of JavaScript's runtime environment. It continuously checks the call stack and other queues to determine if there are any pending tasks. Its main job is to ensure that the call stack is empty before processing the next task. If the call stack is empty, the event loop takes the next task from the callback queue (or the microtask queue) and pushes it onto the call stack for execution.
3. **Callback Queue**: The callback queue, also known as the task queue, is a queue-like structure that holds tasks or functions to be executed. These tasks usually come from asynchronous operations, such as timers, AJAX requests, or event handlers. When an asynchronous task is completed, its corresponding callback function is placed in the callback queue. The event loop checks the callback queue whenever the call stack is empty and moves tasks from the queue to the call stack for execution.
4. **Microtask Queue**: The microtask queue, also called the job queue or microtask queue, is a separate queue that holds microtasks. Microtasks are special tasks that have higher priority than regular tasks in the callback queue. Microtasks typically include promises, mutation observers, and some APIs like queueMicrotask(). When the call stack becomes empty after executing a regular task, the event loop checks the microtask queue first and processes all the microtasks before moving to the callback queue.

Question 10

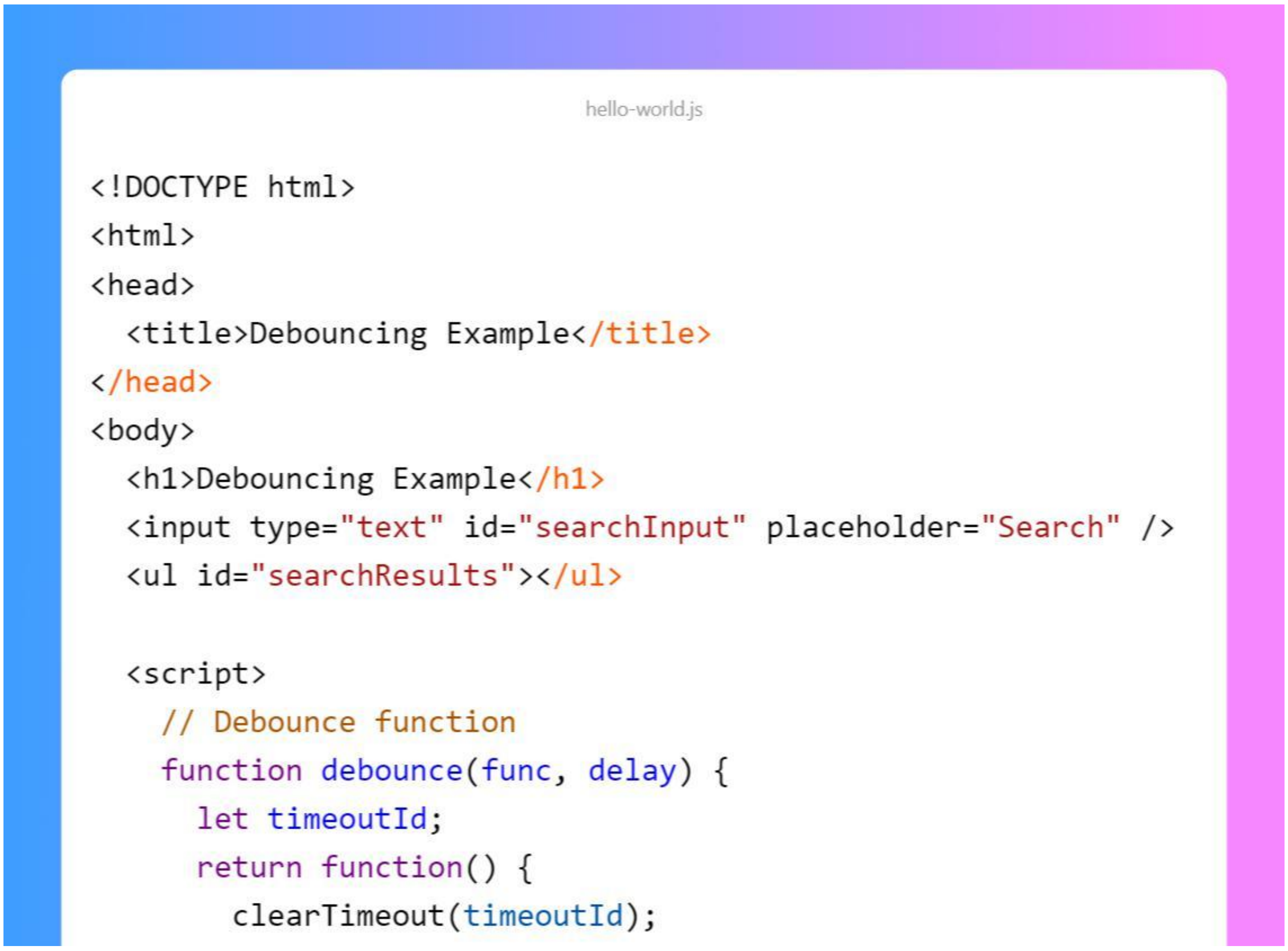
Explain Debouncing and Create a project where you are using Debouncing

Ans:

Debouncing is a technique used in JavaScript to control the frequency of invoking a function based on a specific event. It ensures that the function is executed only after a certain period of inactivity, preventing excessive or unnecessary function calls.

The main purpose of debouncing is to optimize performance by reducing the number of function invocations, especially for events that can occur rapidly or frequently, such as scroll events, resize events, or keystrokes. It helps to delay the execution of a function until a pause occurs, which can be beneficial for resource-intensive operations or scenarios where immediate feedback is not required.

Here's an example project where debouncing is used to improve the performance of an input search functionality:


hello-world.js

```
<!DOCTYPE html>
<html>
<head>
  <title>Debouncing Example</title>
</head>
<body>
  <h1>Debouncing Example</h1>
  <input type="text" id="searchInput" placeholder="Search" />
  <ul id="searchResults"></ul>

  <script>
    // Debounce function
    function debounce(func, delay) {
      let timeoutId;
      return function() {
        clearTimeout(timeoutId);
        timeoutId = setTimeout(func, delay);
      }
    }

    // Perform search operation
    function performSearch() {
      const searchTerm =
document.getElementById('searchInput').value;
      const searchResultsElement =
document.getElementById('searchResults');

      // Simulating an API call with a delay
      setTimeout(() => {
        // Clear previous results
        searchResultsElement.innerHTML = '';

        // Perform search operation and display results
        for (let i = 1; i <= 5; i++) {
          const resultItem = document.createElement('li');
          resultItem.textContent = `Result ${i} for
'${searchTerm}'`;
          searchResultsElement.appendChild(resultItem);
        }
      }, 500);
    }

    // Debounced search function
    const debouncedSearch = debounce(performSearch, 300);

    // Attach event listener to search input
    const searchInput =
document.getElementById('searchInput');
    searchInput.addEventListener('input', debouncedSearch);
  </script>
</body>
</html>
```

In this example, we have an input field for performing a search operation. The performSearch function is responsible for executing the search logic and displaying the results. The debounce function is a utility function that takes a function and a delay time as arguments and returns a debounced version of the function.

When the user types in the search input, the input event triggers. Instead of invoking the performSearch function directly, we use the debouncedSearch function, which is the debounced version of performSearch. The debouncedSearch function delays the execution of performSearch by 300 milliseconds.

As the user types, the debouncedSearch function is invoked repeatedly, but the performSearch function is executed only when there is a pause in typing for 300 milliseconds. This ensures that the search operation is not performed with every keystroke, reducing unnecessary requests and improving performance.

Question 11

Explain Closures and Use cases of Closures

Ans:

Closures are a powerful concept in JavaScript that allow functions to retain access to variables from their parent scope even after the parent function has finished executing. In other words, a closure is a combination of a function and the lexical environment within which that function was declared.

When a function is defined inside another function, the inner function forms a closure that has access to its own local variables, the variables of the outer function, and any global variables. This access is possible even when the outer function has already returned.

Closures are useful in various scenarios, including:

1. **Data Privacy**: Closures provide a way to create private variables and encapsulate data within a function. The inner function can access and modify the variables of its outer function, but those variables are not accessible from the outside. This allows for information hiding and helps in maintaining data integrity.

2. **Function Factories**: Closures enable the creation of function factories, where a function returns another function with some preset configuration or data. The returned function retains access to the variables of its parent function, allowing for customizable behavior.
3. **Asynchronous Operations**: Closures are often used in asynchronous operations, such as event handlers or AJAX requests. They allow the callbacks to access the variables in their surrounding scope even when they are executed at a later time.
4. **Memoization and Caching**: Closures can be used for memoization or caching values. By caching the results of expensive operations, subsequent calls to the function can be avoided if the same input is provided again.

Question 12

Create a Blog web app using JavaScript (10 Marks)
- Fetch data from https://jsonplaceholder.typicode.com/posts and show it to ui
- User can also add new blog
- Add Delete functionality also

GitHub Repo Link:
https://github.com/Pritika17/Placement-Assignment-PritikaMishra-JSQ12

React Questions

Question 1

What's React and What are the advantages of it?

Ans:

React is a popular JavaScript library used for building user interfaces. It was developed by Facebook and released in 2013. React allows developers to create reusable UI components and build interactive and dynamic web applications.

Advantages of React include:

1. Component-based architecture: React follows a component-based approach, where the UI is broken down into reusable components. This promotes code reusability, modularity, and maintainability, making it easier to develop and maintain large-scale applications.
2. Virtual DOM: React uses a virtual representation of the actual DOM (Document Object Model) called the Virtual DOM.

3. Unidirectional data flow: React follows a unidirectional data flow, also known as one-way binding. Data flows in a single direction, from parent components to child components. This simplifies the debugging process and makes the application more predictable.
4. JSX syntax: React uses JSX (JavaScript XML) syntax, which allows developers to write HTML-like code within JavaScript. JSX makes it easier to define the structure and logic of components in a single file, enhancing readability and maintainability.
5. Rich ecosystem and community support: React has a vast ecosystem with numerous third-party libraries, tools, and extensions. This ecosystem provides solutions for various use cases, making it easier to integrate React with other technologies. Additionally, React has a large and active community that continuously contributes to its development, provides support, and shares knowledge through forums, tutorials, and open-source projects.
6. Performance optimization: React's reconciliation algorithm, coupled with its Virtual DOM, helps optimize performance. React efficiently updates and re-renders only the necessary components, reducing unnecessary rendering cycles and enhancing overall application performance.
7. Mobile app development: React Native, a framework built on top of React, enables developers to build native mobile applications using React. By leveraging existing React skills, developers can create cross-platform mobile apps for iOS and Android, saving development time and effort.

Question 2

What's Virtual Dom in React & What are the advantages of it?

Ans:

In React, the Virtual DOM (Document Object Model) is a lightweight copy of the actual DOM. It is a JavaScript representation of the HTML structure of a web page. When there are changes in the application's state or props, React uses the Virtual DOM to efficiently update and re-render only the necessary components in the actual DOM.

Here's how the Virtual DOM works in React:

1. Initial render: When a React component is initially rendered, it creates a Virtual DOM representation of the component's UI structure. This Virtual DOM

is a tree-like structure with nodes representing different components and elements.
2. State or prop changes: When there are changes in the component's state or props, React updates the Virtual DOM instead of directly manipulating the actual DOM.
3. Diffing and reconciliation: After the Virtual DOM is updated, React performs a process called "diffing" to determine the minimal set of changes required to bring the Virtual DOM in sync with the actual DOM. React efficiently compares the previous Virtual DOM with the updated Virtual DOM and identifies the differences.
4. Efficient updates: Once the differences are identified, React applies only those changes to the actual DOM, updating only the affected components and elements. This process of updating the actual DOM is known as reconciliation.

Advantages of the Virtual DOM in React:

1. Performance optimization: The Virtual DOM allows React to optimize the rendering process. Instead of directly manipulating the actual DOM for every change, React updates the Virtual DOM first. By doing so, React reduces the number of direct DOM manipulations, which are typically slower, and performs batch updates to the actual DOM, resulting in better performance.
2. Efficient rendering: With the help of the Virtual DOM, React intelligently determines which components and elements need to be updated. It compares the previous and updated Virtual DOMs to identify the minimal set of changes, reducing the number of re-renders and unnecessary updates. This leads to more efficient rendering and faster UI updates.
3. Cross-platform compatibility: The Virtual DOM abstracts the actual DOM implementation details. This means that React applications built with the Virtual DOM can work consistently across different platforms and browsers. React takes care of managing the differences between various DOM implementations, providing a unified programming model.
4. Development simplicity: React's Virtual DOM simplifies the development process. Developers can work with a JavaScript representation of the UI structure instead of directly manipulating HTML elements. This allows for a more declarative and intuitive approach to building UI components, leading to cleaner and more maintainable code.
5. Testing and debugging: The Virtual DOM makes it easier to test and debug React components. Since the Virtual DOM is a JavaScript object, developers

can inspect and manipulate it, enabling easier debugging and testing of UI components without directly interacting with the actual DOM.

Question 3

Explain LifeCycle of React Components?

Ans:

With the introduction of React Hooks in newer versions of React, the lifecycle of functional components has been enhanced and simplified. Here's an overview of the lifecycle phases and the corresponding hooks:

1. Mounting Phase:
   - useEffect(() => {...}, []): This hook is similar to componentDidMount and is called after the component is mounted to the DOM. The callback function inside useEffect runs only once, similar to the behavior of componentDidMount.
2. Updating Phase:
   - useEffect(() => {...}): This hook is invoked after every render cycle. You can use it to perform side effects or subscribe to changes in state or props. By default, it runs after every render, but you can optimize it by specifying dependencies.
   - useEffect(() => {...}, [dependency]): By providing a dependency array as the second argument to useEffect, you can control when the effect runs. If the dependencies change between renders, the effect will run again.
3. Unmounting Phase:
   - useEffect(() => () => {...}, []): By returning a cleanup function from the effect callback, you can mimic the behavior of componentWillUnmount. The cleanup function is executed when the component is unmounted from the DOM.

Additionally, React Hooks provide other hooks that can be used to manipulate state and control rendering:

- useState(initialState): This hook allows functional components to have their own local state. It returns a state variable and a function to update that state.
- useReducer(reducer, initialState): This hook is an alternative to useState and provides a way to manage complex state logic with reducers.
- useContext(context): This hook allows functional components to consume values from a Context API.

- useMemo(() => {...}, [dependency]): This hook memoizes a value, only recomputing it when the dependencies change.
- useCallback(() => {...}, [dependency]): This hook memoizes a function, preventing unnecessary re-rendering of components that depend on it.

The introduction of React Hooks simplifies the component lifecycle by providing hooks that allow you to control the behavior at each stage of a component's lifecycle, all within a functional component.

Question 4

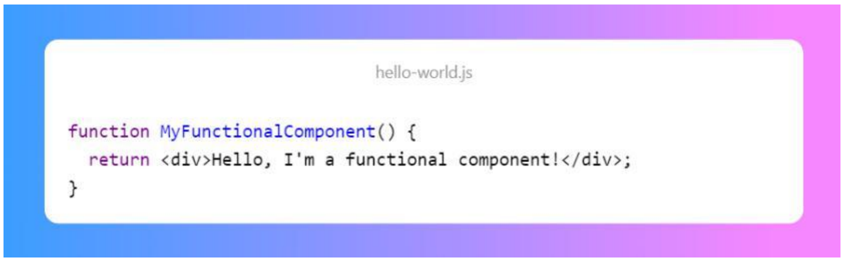
What's the difference between Functional Components and Class Components?

Ans:

Functional components and class components are two ways of creating components in React, a JavaScript library for building user interfaces. Here are the key differences between the two:

1. Syntax: The syntax for creating functional components and class components is different. Functional components are defined as JavaScript functions, while class components are defined as JavaScript classes.

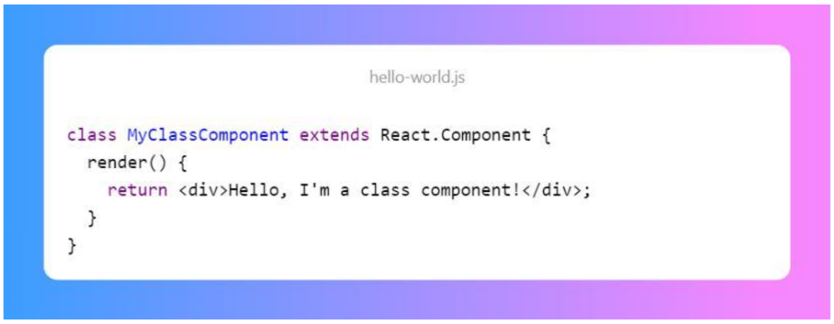
Functional Component Example:



hello-world.js

```
function MyFunctionalComponent() {
  return <div>Hello, I'm a functional component!</div>;
}
```

Class Component Example:



hello-world.js

```
class MyClassComponent extends React.Component {
  render() {
    return <div>Hello, I'm a class component!</div>;
  }
}
```

1. State: Class components have their own internal state, which allows them to manage and update data. Functional components, on the other hand, don't have built-in state. However, with the introduction of React hooks in React 16.8, functional components can now use the useState hook to manage state.
2. Lifecycle Methods: Class components have lifecycle methods, such as componentDidMount, componentDidUpdate, and componentWillUnmount, which allow developers to perform actions at specific points in a component's lifecycle. Functional components, prior to React 16.8, didn't have lifecycle methods. With the introduction of React hooks, functional components can now use the useEffect hook to perform side effects and mimic the behavior of lifecycle methods.
3. Code Organization: Class components tend to have more code and can be more verbose compared to functional components. Functional components are simpler and more lightweight, as they focus solely on rendering UI based on the provided props and state.
4. Performance: Functional components are generally more performant than class components. This is because functional components don't carry the overhead of maintaining an instance and lifecycle methods like class components do. Additionally, functional components can take advantage of React's memoization techniques, such as the React.memo higher-order component, to optimize re-renders.
5. Context and Refs: Class components have their own specific ways of using context and refs, while functional components can use the useContext and useRef hooks to achieve the same functionality.

Question 5

What are the hooks in React & Can we use Hooks in Class Components?

Ans:

In React, hooks are functions that allow you to use state and other React features in functional components. They were introduced in React 16.8 as a way to write reusable logic and manage state in functional components without the need for class components. Hooks provide a more intuitive and concise way to work with React's features.

Here are some commonly used hooks in React:

1. useState: This hook allows functional components to have state. It returns a state value and a function to update that state.
2. useEffect: This hook is used to perform side effects in functional components. It runs after rendering and can be used for tasks such as fetching data, subscribing to events, or updating the DOM.
3. useContext: This hook allows functional components to access the value of a context provided by a Context.Provider higher up in the component tree.
4. useRef: This hook returns a mutable ref object, which can be used to store a value across renders. It is often used to access DOM elements or to store mutable values without triggering re-renders.
5. useReducer: This hook is an alternative to useState for managing more complex state logic. It uses a reducer function to update the state based on dispatched actions.
6. useCallback and useMemo: These hooks optimize the performance of functional components by memoizing values. useCallback memorizes a function, while useMemo memorizes a value.
7. useContext, useReducer, useCallback, and useMemo can also be used together to create custom hooks, encapsulating reusable logic.

Now, regarding your question, hooks are designed to be used in functional components. Class components do not support hooks directly. Hooks rely on the order of execution and the functional nature of components, which is not available in class components.

However, if you have existing class components and want to use hooks, you can use the react-hooks-classes library. It provides a way to use hooks within class components by using a higher-order component (HOC) called withHooks. This approach can be helpful if you want to gradually migrate from class components to functional components with hooks.

That being said, it is generally recommended to use functional components with hooks in new projects or when refactoring existing code. Functional components offer better performance, simpler syntax, and improved reusability compared to class components.

Question 6

What are the LifeCycle methods and the advantages of it?

Ans:

Lifecycle methods in React are special methods that are invoked at specific stages of a component's lifecycle. They allow you to hook into these stages and perform actions or update the component accordingly. Here are the main lifecycle methods in React class components:

1. componentDidMount: This method is called after the component has been rendered to the DOM. It is commonly used for initializing third-party libraries, setting up event listeners, or fetching data from an API.
2. componentDidUpdate: This method is called whenever the component's props or state change, except for the initial render. It allows you to respond to changes in the component and perform actions accordingly. It is often used for updating the DOM based on new data or triggering side effects.
3. componentWillUnmount: This method is called just before the component is removed from the DOM. It is used to clean up any resources that were created in componentDidMount, such as canceling timers, removing event listeners, or unsubscribing from external subscriptions.
4. shouldComponentUpdate: This method is called before the component re-renders. It allows you to control whether the component should update or not based on the new props or state. By default, React will re-render the component whenever there is a change in props or state. Implementing shouldComponentUpdate can optimize performance by preventing unnecessary re-renders.
5. componentDidCatch: This method is called when an error occurs during rendering, in a lifecycle method, or in the constructor of any child component. It is used for error handling and displaying fallback UI when an error occurs.

Advantages of Lifecycle Methods:

1. Control over component behavior: Lifecycle methods provide control and customization over how a component behaves at different stages of its lifecycle. You can perform specific actions or update the component based on the lifecycle events.
2. Initialization and cleanup: Lifecycle methods like componentDidMount and componentWillUnmount allow you to initialize resources or subscriptions when the component is mounted and clean them up when the component is unmounted. This helps avoid memory leaks and ensures proper management of external dependencies.
3. Optimizing performance: Lifecycle methods such as shouldComponentUpdate provide the ability to optimize component rendering. By implementing this method, you can prevent unnecessary re-renders by determining if a component update is required based on the new props or state.
4. Error handling: The componentDidCatch method helps catch and handle errors that occur during rendering or within child components. It allows you to gracefully handle errors and display fallback UI instead of crashing the entire application.

5. Integration with third-party libraries: Lifecycle methods provide integration points for interacting with third-party libraries. You can initialize or teardown external libraries in the appropriate lifecycle methods, ensuring proper synchronization with the component's lifecycle.

It's important to note that with the introduction of React hooks, the use of lifecycle methods has been largely replaced by hooks like useEffect and useLayoutEffect in functional components. Hooks provide a more declarative and composable way to achieve the same functionality while avoiding class component complexities.

Question 7

What’s useState Hook & Advantages of it?

Ans:

The useState hook is a built-in hook in React that allows functional components to have state. It provides a way to add and manage stateful values within functional components without the need for class components. The useState hook returns an array with two elements: the current state value and a function to update that value.

Advantages of the useState hook:

1. Simplicity: The useState hook simplifies state management in functional components. It allows you to define and update state within the component function itself, without the need for a separate class or complex syntax.
2. Reusability: The useState hook is highly reusable. You can use it multiple times within a single component to define and manage multiple state values.
3. Functional approach: The useState hook promotes a functional programming approach by encouraging the use of immutable updates. Instead of directly modifying the state, you use the updater function returned by useState to update the state based on its previous value. This helps in avoiding state mutation and ensures predictable state changes.
4. No class component overhead: With the useState hook, you can use state in functional components without the need for class components. This eliminates the need for class-specific syntax, lifecycle methods, and the overhead of creating a class instance.
5. Performance optimization: The useState hook provides a performance optimization through a mechanism known as "batching." Multiple setState calls within the same render cycle are batched together, resulting in a single state update and a single re-render. This can improve the performance of your application.

6. Integration with other hooks: The useState hook integrates seamlessly with other hooks provided by React, such as useEffect, useContext, and custom hooks. This allows you to combine different hooks and build complex functionality within functional components.
7. Migrating from class components: The useState hook, along with other hooks, provides a way to migrate from class components to functional components. Hooks allow you to achieve the same state management and lifecycle functionality as class components, making it easier to refactor or rewrite code.

Question 8

Explain useEffect & Advantages of it?

Ans:

The useEffect hook is another built-in hook in React that allows functional components to perform side effects. Side effects can include fetching data, subscribing to events, manipulating the DOM, or any other operation that isn't directly related to rendering UI.

The useEffect hook takes two arguments: a callback function that represents the side effect, and an optional array of dependencies. The callback function will be executed after the component renders, and it can return a cleanup function to handle any necessary cleanup before the component is unmounted or re-rendered.

Advantages of the useEffect hook:

1. Side effect management: The useEffect hook provides a declarative way to manage side effects within functional components. By placing side effect code inside the useEffect callback, you can clearly separate the side effect logic from the UI rendering logic.
2. Lifecycle integration: The useEffect hook can replicate the functionality of lifecycle methods in class components. By specifying dependencies in the dependency array, you can control when the effect runs (on mount, on specific prop or state changes, or on unmount). This allows you to perform actions at specific points in the component's lifecycle.
3. Asynchronous operations: The useEffect hook is ideal for handling asynchronous operations, such as fetching data from an API. You can use asynchronous functions or promises within the effect callback to perform the asynchronous task, update the state, and trigger UI re-renders accordingly.
4. Dependency tracking: The optional dependency array in the useEffect hook allows you to specify dependencies that the effect relies on. React will compare the previous and current dependencies and only re-run the effect if they have changed.

This helps optimize performance by preventing unnecessary re-execution of the effect.

5. Cleanup: The useEffect hook supports cleanup operations via the optional cleanup function. This function is called when the component is unmounted or before the effect is re-executed. It can be used to cancel subscriptions, clear timers, remove event listeners, or perform any other necessary cleanup tasks.

Question 9

Explain Context Api and create a minor project on it 0

- Create dashboard and with button on clicking on that change theme to dark and light

Ans:

The Context API is a feature provided by React that allows you to manage global state in your application without the need for prop drilling (passing props through multiple components). It provides a way to share data between components without explicitly passing the data down through the component tree.

GitHub Repo Link:
[https://github.com/Pritika17/Placement-Assignment-PritikaMishra-React Q9](https://github.com/Pritika17/Placement-Assignment-PritikaMishra-React)

Question 10

Explain useReducer and Its advantages.

Ans:

useReducer is a React hook that provides an alternative way to manage state in components. It accepts a reducer function and initial state, returning the current state and a dispatch function. Its advantages include:

1. Simplifying complex state management by encapsulating logic in a reducer function.
2. Enabling predictable state transitions through defined actions.

3. Facilit
4. ating shared state between components, aiding testing, debugging, and potential performance optimizations.

Question 11
build a Todo Web App Using React and useReducer Hook.

GitHub Repo Link:
https://github.com/Pritika17/Placement-Assignment-PritikaMishra-ReactQ11

Question 12
Build A simple counter app using React

GitHub Repo Link:
https://github.com/Pritika17/Placement-Assignment-PritikaMishra-ReactQ12

Question 13
Build Calculator Using React Only

GitHub Repo Link:
https://github.com/Pritika17/Placement-Assignment-PritikaMishra-ReactQ13

Question 14
Build a Tic Tac Toe Game using Class Component of React

GitHub Repo Link:
https://github.com/Pritika17/Placement-Assignment-PritikaMishra-ReactQ14

Question 15
Explain Prop Drilling & How can we avoid it?

Ans:

Prop drilling is a term used in React to describe the process of passing props down multiple levels of nested components, even when some of the intermediate components do not directly use or require those props. It can occur when a component needs to pass data to its child components, which in turn pass the data to their child components, and so on. This can lead to unnecessary and cumbersome code, as well as decreased code maintainability.

Prop drilling can be problematic for the following reasons:

1. Complexity: As the application grows and the component hierarchy deepens, prop drilling can make it difficult to understand how data is being passed between components. Developers have to trace the flow of props through multiple layers, which can be time-consuming and error-prone.

2. Readability: Prop drilling adds extra code and noise to the components that do not need the props, making the code less readable and harder to maintain.

3. Coupling: Prop drilling creates a tight coupling between components, as intermediate components become dependent on passing props to child components that may not be directly related. This can make it challenging to refactor or modify components in the future.

To avoid prop drilling, we can employ the following techniques:

1. Context API: The Context API allows us to share data across multiple components without the need for explicit prop passing. We can create a context at a higher level in the component tree and provide the data to the components that need it. This way, we can avoid passing props through intermediate components that do not require them.
Context provides a way to access the shared data without explicitly passing props at every level.

2. Redux: Redux is a state management library that helps manage global state in a predictable manner. By centralizing the state in a store, components can access the required data without the need for prop drilling. Redux provides a way to connect components to the store and access the required data using selectors.

3. React Router: If prop drilling is mainly required for routing-related data, React Router can be used to manage the routing state. React Router provides a way to access the router's history, location, and match information without passing them explicitly through props.

4. Component Composition: Rather than passing data through multiple layers of components, consider restructuring the component hierarchy to improve component composition. By breaking down components into smaller, more focused components, we can minimize the need for prop drilling.

Express Questions

Question 1

Create a simple server using Express and connect with backend and create an endpoint "/post" which sends 20 posts

GitHub Repo Link:
https://github.com/Pritika17/Placement-Assignment-PritikaMishra-ExpressQ1

Question 2

Explain a middleware and create a middleware that checks is user authenticated or not then send data of post

Ans:

Middleware functions are an essential part of many web frameworks, including Express. Middleware functions are functions that have access to the request (req) and response (res) objects in an Express application's request-response cycle. They can perform various tasks such as modifying the request or response objects, executing additional code, or terminating the request-response cycle.

GitHub Repo Link:
https://github.com/Pritika17/Placement-Assignment-PritikaMishra-ExpressQ2

Question 3

Create a backend for blog app, where user can perform crud operations
- Add blog
- Delete blog
- Update blog
- Replace blog

GitHub Repo Link:
https://github.com/Pritika17/Placement-Assignment-PritikaMishra-ExpressQ3

Question 4

What is the difference between authentication and authorization?

Ans:

Authentication and authorization are two distinct concepts related to controlling access and ensuring security in a system. Although they are often used together, they serve different purposes:

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. The goal of authentication is to establish trust and grant access to the system or resources based on the provided credentials. Authentication commonly involves presenting a username and password combination, but it can also involve other factors such as biometric data, security tokens, or digital certificates. Once authenticated, the user is granted a form of identification (e.g., a session token or a cookie) to represent their authenticated state during subsequent interactions.

Authorization:

Authorization, on the other hand, deals with granting or denying permissions to authenticated users. It determines what actions or resources a user is allowed to access or perform within a system. Authorization is based on the user's role, privileges, or specific permissions associated with their authenticated identity. It ensures that users only have access to the parts of the system that they are allowed to use. Authorization is typically enforced through access control mechanisms, such as role-based access control (RBAC) or attribute-based access control (ABAC)

Question 5:0
What is the difference between common JS and EJS module?

Ans:

The main difference between CommonJS (CJS) and EJS (Embedded JavaScript) modules lies in their module systems and usage patterns:

1. Module System:
   - CommonJS (CJS): CJS is the module system used in Node.js. It follows a synchronous, "require/import and export" approach. Modules are loaded and executed synchronously, and dependencies are resolved at runtime.
   - EJS (Embedded JavaScript): EJS is a templating language commonly used in web development. It allows embedding JavaScript code within HTML templates. EJS does not have its own module system but can be used in conjunction with other module systems like CommonJS or ES modules.
2. Usage Pattern:
   - CommonJS (CJS): In CJS modules, dependencies are imported using the require function, and modules are exported using the module.exports or exports objects. CJS modules are typically used in server-side JavaScript, where synchronous loading of dependencies is common.
   - EJS (Embedded JavaScript): EJS is primarily used as a templating engine for generating dynamic HTML content. It allows embedding JavaScript code within HTML templates using tags like <% %>, <%= %>, and <%- %>. EJS templates are usually compiled to HTML with the help of a rendering engine.

Question 6
What is JWT and what we can achieve with that create a minor project with jwt

Ans:
JWT stands for JSON Web Token. It is an open standard (RFC 7519) for securely transmitting information between parties as a JSON object. JWTs are commonly used for authentication and authorization in web applications.
A JWT consists of three parts: header, payload, and signature, separated by dots (.). The header specifies the algorithm used for signing the token, while the payload contains the claims (statements) about the user or any other data. The signature is generated by combining the encoded header, payload, and a secret key using the specified algorithm.

GitHub Repo Link:
https://github.com/Pritika17/Placement-Assignment-PritikaMishra-ExpressQ6

Question 7:0

What should we do with the password of a user before storing it into DB?

Ans:

When storing a user's password in a database, it is essential to follow security best practices to protect the user's sensitive information. Here are some recommended steps to handle the password before storing it:

1. Hashing: Instead of storing the plain text password, always hash the password using a secure one-way hashing algorithm. Hashing is a process of converting the password into a fixed-length string representation that cannot be reversed to obtain the original password.
2. Salt: Use a unique salt value for each user's password before hashing. A salt is a random value added to the password before hashing, which ensures that even if two users have the same password, their hashed passwords will be different.
   Salting strengthens the security of hashed passwords and mitigates against precomputed rainbow table attacks.
3. Strong Hashing Algorithm: Choose a strong hashing algorithm specifically designed for password hashing, such as bcrypt, Argon2, or scrypt. These algorithms are purposely slow and computationally expensive, making it more difficult for attackers to crack the hashed passwords through brute-force or dictionary attacks.
4. Iterations/Work Factor: Configure the hashing algorithm to use a sufficient number of iterations or work factor. Increasing the number of iterations or work factor slows down the hashing process, which makes it harder for an attacker to perform brute-force or dictionary attacks.
5. Store Hashed Password: Only store the resulting hashed password in the database, not the plain text password or the salt. The hashed password alone is sufficient for password verification during the authentication process

Question 8

What's the event loop in NodeJS?

Ans:

The event loop is the core mechanism in JavaScript that allows non-blocking, asynchronous execution of code. It is responsible for managing and processing events, callbacks, and I/O operations.

Here's a simplified explanation of how the event loop works:

1. JavaScript code is executed in a single-threaded environment.
2. When an asynchronous operation (such as an API request or a timer) is encountered, it is registered and the execution continues to the next statement.
3. Once the asynchronous operation is completed, a callback associated with that operation is added to a task queue.
4. The event loop continuously checks for any completed tasks in the task queue.
5. If a task is found, the associated callback is pushed onto the execution stack for processing.
6. The callback is executed, and if there are any additional asynchronous operations within it, the process is repeated.
7. This cycle of checking the task queue, executing callbacks, and continuing with the event loop continues until there are no more tasks in the queue.

The event loop allows JavaScript to handle multiple concurrent operations without blocking the main thread. It ensures that callbacks are executed in the order they were added to the task queue, maintaining the asynchronous and non-blocking nature of JavaScript.

It's important to note that the event loop operates differently in different environments. For example, in Node.js, there are additional phases in the event loop related to I/O operations, while in the browser, there are phases related to rendering and user interactions.

Question 9

Create a Full Stack Ecommerce website with all major functionalities.

GitHub Repo Link:
https://github.com/Pritika17/Placement-Assignment-PritikaMishra-ExpressQ9