**ASSIGNMENT**

**HTML Answers**

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

* <DOCTYPE html> is not a tag in html, it is known as “**Document Type Declaration**” or “**Document Type Definition**” (DTD).
* It is used to define the type or version of html being used in webpage.
* <DOCTYPE html> declaration is used in html to indicate that the document is written in HTML syntax.
* The purpose of <DOCTYPE html> is to inform the browser about the version of html document, which helps the browser to render the page correctly.
* It helps to ensure better cross-browser compatibility and consistent rendering of webpages across different devices and platforms.

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

* Semantic tags in HTML are elements that provide meaning and structure to the content within a webpage.
* Unlike generic tags **“<div>”** and **“<span>”**. Semantic tags describe the purpose and role of the content they wrap.
* They help both humans and search engines to understand the significance of different sections of a webpage.

Here are some examples of semantic tags introduced in HTML5: -

**<header>,<nav>, <main>,<article>,<section>,<aside>, <footer>**

In summary, semantic tags in HTML add meaning and structure to webpage content improving accessibility, search engine optimization and forward compatibility.

1. Differentiate between HTML Tags and Elements?

**Tags –** Tags are fundamental building blocks of HTML syntax. They are used to define the structure and appearance of elements within HTML document. Tags are represented by angle brackets **“<”** and **“>”** and usually comes in pairs with an opening tag and closing tag.

Ex-

**<p> ………. </p>** is an example of tag. **<p>** is the known as opening tag and **</p>** is known as closing tag.

It is important to note that some tags are void or empty don’t having closing tag and are self-closed with forward slash **( ‘/ ‘)** before the closing angle bracket of the opening tag for example,

<img src=”image.png” alt=”image” />, <br/> and <hr/> etc.

**Elements –** Elements are composed of tags, along with the content or other element they enclose. An element consists of an opening tag with content and a closing tag. The opening tag marks the beginning of the element and closing tag marks the end of the element. The content be text or other elements or a combination of both.

Ex –

**<p> This is paragraph</p>** as a whole is an example of element where **<p>** is opening tag **“This is paragraph”** is content and **</p>** is closing tag.

1. Build Your Resume using HTML only.

Repo Link - <https://github.com/Gaurav-Dev24/Placement-Assignment/blob/main/HTML%20Answers/Q.no_4_resume.html>

1. Write Html code so that it looks like the given image Link

Repo Link - <https://github.com/Gaurav-Dev24/Placement-Assignment/blob/main/HTML%20Answers/Q.no_5_image.html>

1. What are some of the advantages of HTML5 over its previous versions?

HTML5 is the fifth revision of Hyper Text Mark-up Language, which has been introduced with several significant advancements over previous versions. Some of the major changes are as followed:-

* **Rich Multimedia Support**
* **Improved Backward Compatibility**
* **Enhanced New Semantics such as <header>, <footer>, <article>, <section>**
* **Improved form elements**
* **Introduction of localStorage API**

1. Create a simple Music player using html only

Repo Link - <https://github.com/Gaurav-Dev24/Placement-Assignment/blob/main/HTML%20Answers/Q.no_7_music_player.html>

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

The major difference between <img> and <figure> is <img> tag is used to embed image in html document. It is self-closing tag that require src attributes to specify the image source or URL.

Whereas <figure> tag is used to group and represent self-contained content such as images, diagrams and code snippets. It provides a way to associate caption or description with the content it encloses.

Apart from it <img> tag is primarily used for displaying image it does not have any additional semantic meaning whereas in <figure> tag, it provides semantic meaning to the content.

Syntax of image tag - <img src=”./images/img.png” alt=”image”/>

Syntax for figure tag –

<figure>

<img src=”./images/img.png” alt=”image”/>

<figcaption>Caption</figcaption>

</figure>

1. What’s the difference between html tag and attribute and give example of some global attributes?

Html Tag- An HTML tag represents an element in an HTML document and is enclosed within angle brackets. Tags define the structure and content of the document. They can be categorising into two types “Opening tag” and “Closing tag”. The opening tag denotes the start and closing tag denotes the end of the element. Some elements known as “void” and “empty” does not require closing tag hence known as self-closing element.

Some examples of tags are:

**<p>This is paragraph tag</p>**

**<a href=”#” >This is anchor tag</a**>

<img src=”./images/img.png” alt=”image”/>

Attributes- An attribute is provided in HTML tag to modify or to provide additional information about the element. Attributes are placed inside the opening tag and consist of name-value pair. The attribute name is followed by “=” sign and attribute value is followed by “ “ and ‘ ‘ quotes.

Some of the examples of attributes are;

href, type, target, placeholder src, alt, width and height etc.

<a href="https://www.example.com" target="\_blank">Link</a>

<input type="text" id="username" placeholder="Enter your name">

<img src="image.jpg" alt="Image description" width="500" height="300">

Examples of global attributes are:

* Class – specifies one or more classes to an element
* Id – provides unique identifier to the element
* Style – defines inline css style to the element
* Title – specifies the title or tooltip text for element
* Lang – defines the language of the elements content
* Data – allows custom data attribute to be added to element

1. Build Table which looks like the given image Link

Repo Link - <https://github.com/Gaurav-Dev24/Placement-Assignment/blob/main/HTML%20Answers/Q.no_10_table.html>

**CSS Answers**

1. What is Box Model in CSS & Which CSS Properties is part of it?

The Box model in css describes the layout and sizing of elements on a web page. It consist of four components which defines the total space occupied by an element: content, padding, border, margin.

The properties associated with box model are:

Content –

* Width – specifies the width of the content area
* Height – specifies the height of the content area

Padding –

* top, right, bottom and left determines the space between the content area and border of element.

Border –

* border-width – sets the thickness of border
* border-style – specifies the style of the border (e.g., solid, dashed etc.)
* border-color – specifies the color of the border
* border-radius – specifies the rounded corners of the border.

Margin –

* top, right, bottom and left determines the space between elements border and neighbouring elements.

Together all these properties controls the overall size and spacing of elements in web page.

e.g.-

.box {

width: 200px;

height: 150px;

padding: 10px;

border: 1px solid black;

margin: 20px;

}

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

There are different types of selectors present in CSS to target specific HTML elements and applying css to them. Some of the commonly used selectors are as followed:

**Element Selector** – selects elements based on their element name.

**Eg.**

p {

/\* Styles applied to all <p> elements \*/

}

**Advantages** – simple and straight forward, targets all the element of specific type.

**Class Selectors** – selects elements based on the value of their “class” attribute. Starts with (.) followed by name of the class.

**Eg**.

.highlight {

/\* Styles applied to elements with class="highlight" \*/

}

**Advantages** – allows the grouping and styling of multiple elements across different tags with same class name promoting reusability and maintaining consistency.

**ID Selectors** – selects elements based on the unique value of their “id” attribute. Starts with (#) followed by name of the id.

**Eg**.

. #header {

/\* Styles applied to the element with id="header" \*/

}

**Advantages** – target a single unique element, useful for applying specific styles or JavaScript interactions.

**Attribute Selectors** – selects elements based on their attribute value.

**Eg**.

input[type="text"] {

/\* Styles applied to <input> elements with type="text" \*/

}

**Advantages** –allows targeting element based on the value of attribute providing flexibility and granularity in selection.

**Descendent Selectors** – selects elements that are descendants of another element.

**Eg**.

.container p {

/\* Styles applied to <p> elements within an element with class="container" \*/

}

**Advantages** –enables targeting specific elements within a particular parent container, facilitating contextual styling.

**Child Selectors** – selects direct child of an element.

**Eg**.

ul > li {

/\* Styles applied to <li> elements that are direct children of <ul> \*/

}

**Advantages** –targets immediate child elements, useful for styling the specific level of nested elements.

**Pseudo-classes and Pseudo-elements Selectors** – select the elements based on their state or position.

**Eg**.

a:hover {

/\* Styles applied to <a> elements on hover \*/

}

p::first-letter {

/\* Styles applied to the first letter of <p> elements \*/

}

**Advantages** – enables styling based on dynamic states (e.g. hover, active) or applying styles to specific parts of an element (e.g. first letter, first line).

1. What is VW/VH & How it’s different from PX?

VW (Viewport Width) and VH (Viewport Height) are units of measurement in CSS that are relative to the size of viewport or the browser window.

VW – 1vw = 1% of the width of viewport.

e.g.

If the width of the viewport is 1000px then 1vw is equal to 10 pixels i.e. 1% \* 1000px.

VH – 1vh = 1% of the height of viewport.

e.g.

If the height of the viewport is 800px then 1vw is equal to 8 pixels i.e. 1% \* 800px.

In summary the major difference between the VW/VH and PX is how they are calculated and their responsiveness o changes in viewport size. VW/VH are particularly useful for creating responsive designs that dynamically adapt to the different viewport size, whereas PX provide a fixed size regardless the viewport dimensions.

1. What’s difference between Inline, Inline Block and block?

The difference between “inline”, “inline-block” and “block” in css determines that how elements are rendered and how they interact with other elements within the document flow.

**Inline:** Inline elements do not start on a new line and only take up as much width as necessary to contain their content. They flow alongside each other horizontally within the same line.

Examples of inline elements: <span>, <a>, <strong>, <em>, <img>.

**Inline Block:** Inline-block elements are similar to inline elements in that they flow within the same line horizontally, but they can have a specified width, height, margins, and padding. They retain their block-level properties while staying inline.

Examples of inline-block elements: <input>, <button>, <select>, <div> (with display: inline-block;).

**Block:** Block-level elements start on a new line and take up the full width available, extending from the left edge to the right edge of their containing element by default. They create a block-level formatting context and can contain other block-level and inline elements.

Examples of block-level elements: <div>, <p>, <h1>-<h6>, <ul>, <li>, <section>, <header>, <footer>, etc.

1. How is Border-box different from Content Box?

Content Box:

* The "content-box" is the default box-sizing value in CSS.
* When using "content-box", the specified width and height of an element only include the content area, excluding the padding, border, and margins.
* In other words, the width and height you set for an element using "content-box" represent the dimensions of the content area alone, and the padding, border, and margins are added to it.

Border Box:

* When using "border-box", the specified width and height of an element include the content area, padding, and border.
* In "border-box" sizing, the padding and border are included within the specified width and height, and the browser adjusts the content box size to fit.

The choice between "content-box" and "border-box" depends on the desired box-sizing behaviour. "border-box" is often favoured for its convenience, as it allows easier control over the overall dimensions of elements, especially when dealing with complex layouts and responsive designs.

1. What’s z-index and how does it function?

"z-index" property appropriately allows developers to control the layering and visibility of elements on the web page. It is commonly used in scenarios where elements need to overlap or when creating complex layouts with overlapping components, dropdown menus, or modal dialogs. It's important to note that the "z-index" property affects only elements that have a position value other than "static".

1. What’s Grid & Flex and difference between them?

Grid and Flex box are two popular CSS layout systems that provide different approaches to creating responsive and flexible web page layouts.

**Flex box (Flexible Box Layout):**

Flex box is a one-dimensional layout system that allows you to arrange elements along a single axis (either horizontally or vertically). It provides powerful alignment and distribution capabilities for creating flexible and dynamic layouts.

**Key points about Flex box include:** one dimensional layout, flex containers& flex items, flexibility & alignment.

**Grid:**

Grid is a two-dimensional layout system that allows you to create complex grid-based layouts with rows and columns. It provides precise control over both horizontal and vertical axes, enabling you to create responsive layouts that adapt to different screen sizes.

**Key points about Grid include:** two dimensional layouts, grid containers & grid items, rows & columns and grid tracks & grid areas

Difference between Grid and Flex box:

1. **Layout system:** Flex box is a one-dimensional layout system, while Grid is a two-dimensional layout system.
2. **Axis control:** Flex box works along a single axis (horizontal or vertical), whereas Grid allows control over both the horizontal and vertical axes.
3. **Alignment:** Flex box provides powerful alignment properties for flex items, allowing you to control their positioning within a flex container. Grid offers more advanced alignment capabilities, including alignment of both rows and columns.
4. **Layout complexity:** Grid is more suitable for complex grid-based layouts with multiple rows and columns, whereas Flex box is well-suited for simpler one-dimensional layouts or aligning items within a container.
5. Difference between absolute and relative and sticky and fixed position explain with example.

The differences between absolute and relative positioning, as well as sticky and fixed positioning, are related to how elements are positioned within the document flow and their behaviour when scrolling.

**Absolute Positioning:**

* Absolute positioning removes an element from the normal document flow and positions it relative to its nearest positioned ancestor or the containing block.
* The element's position is specified using the **top, right, bottom,** and **left** properties.
* Other elements are not affected by the absolute positioned element, and it may overlap or be overlapped by other elements.
* If no positioned ancestor is found, the element will be positioned relative to the initial containing block, which is usually the viewport.

e.g.

**<div class="container">**

**<div class="absolute-box"></div>**

**</div>**

**.container {**

**position: relative;**

**height: 300px;**

**}**

**.absolute-box {**

**position: absolute;**

**top: 50px;**

**left: 50px;**

**width: 200px;**

**height: 200px;**

**background-color: red;**

**}**

In the above example, the **.absolute-box** element is positioned absolutely within its relative positioned parent container. It is placed **50 pixels** from the top and **50 pixels** from the left of its nearest positioned ancestor.

**Relative Positioning:**

* Relative positioning allows an element to be positioned relative to its normal position in the document flow.
* The element's position is adjusted using the **top, right, bottom,** and **left** properties, which shift it from its original position.
* Other elements on the page are not affected by the relative positioned element, and it still occupies its original space in the document flow.

e.g.

**<div class="relative-box"></div>**

**.relative-box {**

**position: relative;**

**top: 20px;**

**left: 30px;**

**width: 150px;**

**height: 150px;**

**background-color: blue;**

**}**

In the above example, the **.relative-box** element is positioned relative to its original position. It is shifted **20 pixels down** and **30 pixels to the right** from where it would normally appear.

**Sticky Positioning:**

* Sticky positioning is a combination of both relative and fixed positioning.
* It behaves like a relatively positioned element until it reaches a specific scroll position, after which it becomes fixed in place.
* The element's position is determined using the **top, right, bottom,** and **left** properties when it becomes sticky.
* Sticky positioning is commonly used for creating sticky headers, navigation menus, or sidebars that stay fixed until a certain point while scrolling.

e.g.

**<div class="sticky-box"></div>**

**.sticky-box {**

**position: sticky;**

**top: 20px;**

**width: 200px;**

**height: 100px;**

**background-color: green;**

**}**

In the above example, the **.sticky-box** element starts as a relatively positioned element. However, when the user scrolls to a specific point (defined by **top: 20px;**), it becomes fixed in place and remains visible even as the rest of the content scrolls.

**Fixed Positioning:**

* Fixed positioning removes an element from the normal document flow and positions it relative to the viewport (the browser window).
* The element's position is specified using

1. Build Periodic Table as shown in the below image.

Repo Link - <https://github.com/Gaurav-Dev24/Placement-Assignment/tree/main/CSS%20Answers/Question%209>

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

Repo Link - <https://github.com/Gaurav-Dev24/Placement-Assignment/tree/main/CSS%20Answers/Question%2010>

1. Build Responsive Layout both desktop and mobile and Tablet, see below image for reference?

Repo Link - <https://github.com/Gaurav-Dev24/Placement-Assignment/tree/main/CSS%20Answers/Question%2011>

1. What are Pseudo class in CSS & how it’s different From Pseudo Elements?

In CSS, pseudo-classes and pseudo-elements are used to style specific elements based on various conditions or states. While they may sound similar, they have distinct purposes and functionality.

**Pseudo-classes:**

Pseudo-classes target elements based on their current state or relation to the document structure, such as user interaction or element position. **They are denoted by a single colon ":" followed by the name of the pseudo-class.** Some common examples of pseudo-classes are:

**:hover -** Styles an element when the user hovers over it.

**:active -** Styles an element when it is being activated or clicked.

**:focus -** Styles an element when it has received focus.

**:nth-child(n) -** Styles elements that are the nth child of their parent.

**:first-child -** Styles the first child element of its parent.

**:last-child -** Styles the last child element of its parent.

**Pseudo-elements:**

Pseudo-elements, on the other hand, target and style specific parts of an element, allowing you to create additional elements within the selected element. **They are denoted by two colons "::" followed by the name of the pseudo-element.** Some commonly used pseudo-elements include:

**::before -** Inserts content before the selected element.

**::after -** Inserts content after the selected element.

**::first-line -** Styles the first line of the selected element's text.

**::first-letter -** Styles the first letter of the selected element's text.

**::selection** - Styles the portion of text selected by the user.

One key difference between pseudo-classes and pseudo-elements is that pseudo-classes target and style entire elements based on their state or position, while pseudo-elements allow you to create and style specific parts within an element. Pseudo-classes are denoted by a single colon ":" while pseudo-elements are denoted by two colons "::" (although some older pseudo-elements still use a single colon).

It's important to note that not all pseudo-classes and pseudo-elements are supported in all browsers, so it's essential to check their compatibility before using them in your CSS styles.

**JavaScript Answers**

* 1. What is Hoisting in JavaScript?

Hoisting is behaviour in JavaScript where variable and function declarations are moved to the top of their containing scope during the compilation phase, regardless of where they are actually written in the code. This means that you can use variables and call functions before they are declared in your code, and they will still work as expected.

However, it's important to understand that hoisting doesn't move the actual initialization or assignment of variables, only the declaration. Let's look at some examples to illustrate this concept:

**Example 1: Variable Hoisting**

Repo Link – <https://github.com/Gaurav-Dev24/Placement-Assignment/blob/main/JS%20Answers/Q%201/script.js>

In this example, even though the variable `x` is declared later in the code, the `console.log` statement doesn't throw an error. Instead, it prints `undefined`. This is because the declaration of `x` is hoisted to the top, but the assignment `x = 5` is not hoisted. So, at the time of the `console.log` statement, `x` exists but hasn't been assigned a value yet.

**Example 2: Function Hoisting**

Repo Link – <https://github.com/Gaurav-Dev24/Placement-Assignment/blob/main/JS%20Answers/Q%201/script.js>

In this example, the `sayHello()` function is called before its declaration. Still, the code executes without any errors and prints `"Hello!"`. This is because function declarations are fully hoisted to the top of their containing scope, allowing you to call them anywhere in the code.

**It's important to note that hoisting only applies to functions declarations and variable declarations using the `var` keyword.**

Variables declared with `let` and `const` are hoisted as well, but they are not initialized until their actual declaration. This behavior is known as the "temporal dead zone" and helps catch potential errors caused by accessing variables before they are declared.

**Additionally, using strict mode (`"use strict"`) in JavaScript disables the automatic hoisting of variables, making it easier to catch and prevent certain types of errors.**

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

In JavaScript, higher-order functions are functions that can accept other functions as arguments or return functions as results. They provide a powerful way to work with functions as values and enable functional programming paradigms. Some common higher-order functions in JavaScript include:

**map(), reduce(), filter(), setTimeout() and setInterval()** etc.

1. map(): The `map()` method is used to iterate over an array and transform each element based on a provided callback function. It returns a new array with the same length as the original array, where each element is the result of the callback function. The original array remains unchanged. For example:

2. forEach(): The `forEach()` method allows you to iterate over an array and perform an action on each element using a provided callback function. It does not return a new array and instead operates on the original array directly. The purpose of `forEach()` is typically to execute some code for its side effects. For example:

Repo Link – <https://github.com/Gaurav-Dev24/Placement-Assignment/blob/main/JS%20Answers/Q%202/script.js>

The main difference between `map()` and `forEach()` lies in their return values and purposes. It's worth noting that both `map()` and `forEach()` iterate over array elements in order, from index 0 to the last index, and provide the current element, index, and the entire array as arguments to the callback function.

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

In JavaScript, `call()`, `apply()`, and `bind()` are methods that allow you to control the execution context of a function and explicitly set the value of `this` within the function. While they serve a similar purpose, there are some differences in how they are used.

**call():** The `call()` method is used to invoke a function and explicitly specify the value of `this`. It also allows you to pass arguments to the function individually, separated by commas. Here's an example:

Repo Link – <https://github.com/Gaurav-Dev24/Placement-Assignment/blob/main/JS%20Answers/Q%203/call.js>

In the example above, the `call()` method is used on the `greet` function of the `person` object. By passing `anotherPerson` as the first argument, we set `this` within the `greet` function to `anotherPerson`, effectively accessing the `name` property of `anotherPerson`.

**apply():** The `apply()` method is similar to `call()`, but it accepts arguments as an array or an array-like object instead of individually. This can be useful when you have an array of arguments or want to use the `arguments` object. Here's an example:

Repo Link – <https://github.com/Gaurav-Dev24/Placement-Assignment/blob/main/JS%20Answers/Q%203/apply.js>

In this example, the `apply()` method is used similarly to `call()`, but the arguments are passed as an array (`['Hello']`). The effect is the same as in the previous example.

**bind():** The `bind()` method is used to create a new function that, when called, has a specific `this` value set and optionally pre-set arguments. Unlike `call()` and `apply()`, `bind()` does not immediately invoke the function. Instead, it returns a new function with the bound context and arguments. Here's an example:

Repo Link – <https://github.com/Gaurav-Dev24/Placement-Assignment/blob/main/JS%20Answers/Q%203/bind.js>

In this example, `bind()` is used to create a new function `greetJane` that has its `this` value permanently bound to `anotherPerson`. When `greetJane('Hello')` is called, it executes with the bound context, resulting in the expected output.

To summarize:

- `call()` and `apply()` are used to invoke a function with a specific context (`this`) and arguments.

- `call()` accepts arguments individually, while `apply()` accepts an array or array-like object.

- `bind()` creates a new function with a bound context and optional pre-set arguments, but it does not invoke the function immediately.

These methods provide flexibility in controlling the value of `this` and passing arguments when invoking functions. The choice between them depends on the specific requirements and use case of your code.

* 1. Explain Event bubbling and Event Capturing in JavaScript with suitable examples.

Event bubbling and event capturing are two different mechanisms that describe how events propagate through the DOM (Document Object Model) in JavaScript. They determine the order in which event handlers are executed when an event occurs on a nested element.

**Event Bubbling:**

In event bubbling, when an event is triggered on an element, it first executes the event handlers on the innermost element and then moves up the DOM hierarchy, executing handlers on each ancestor element. This propagation resembles bubbles rising to the surface of water.

Here's an example to illustrate event bubbling:

Repo Link – <https://github.com/Gaurav-Dev24/Placement-Assignment/blob/main/JS%20Answers/Q%204/eventBubbling.html>

When the button is clicked, the event bubbles up through the inner div to the outer div. The event handlers are executed in the order of ancestor elements:

Output:

```

Button clicked

Inner div clicked

Outer div clicked

```

In this example, you can see that the event handler on the button is executed first, followed by the event handler on the inner div, and finally, the event handler on the outer div.

**Event Capturing:**

In event capturing, the event is first captured at the outermost element and then propagates down the DOM hierarchy until reaching the target element. This is the opposite direction of event bubbling.

To use event capturing, you need to set the third parameter of the `addEventListener()` method to `true`.

Repo Link – <https://github.com/Gaurav-Dev24/Placement-Assignment/blob/main/JS%20Answers/Q%204/eventCapturing.html>

Output:

```

Outer div clicked

Inner div clicked

Button clicked

```

In this example, when the button is clicked, the event is first captured by the outer div event handler, then the inner div event handler, and finally the button event handler.

**It's important to note that by default, event handling in JavaScript follows the event bubbling mechanism.** Event capturing is less commonly used but can be helpful in specific scenarios where you need to handle events at the outermost level before they reach the target element.

You can choose between event bubbling and event capturing based on your specific needs and the desired order of event execution in your application.

* 1. What is function currying with example?

Function currying is a technique in JavaScript where a function with multiple arguments is transformed into a sequence of functions, each taking a single argument. It allows you to partially apply arguments to a function and create new functions with the remaining arguments.

Here's an example to demonstrate function currying:

Repo Link – <https://github.com/Gaurav-Dev24/Placement-Assignment/blob/main/JS%20Answers/Q%205/currying.js>

In this example, the `multiply()` function takes an argument `a` and returns an inner function that takes another argument `b`. When you invoke `multiply()` with the first argument, it returns the inner function that remembers the value of `a`.

By assigning the result of `multiply(2)` to `multiplyByTwo`, you create a new function that multiplies its argument by 2. So when you call `multiplyByTwo(5)`, it multiplies 5 by 2 and returns the result 10.

Function currying enables you to create specialized functions by pre-filling some arguments. You can then reuse these specialized functions in different contexts or with different remaining arguments.

Here's another example using currying with multiple arguments:

Repo Link – <https://github.com/Gaurav-Dev24/Placement-Assignment/blob/main/JS%20Answers/Q%205/curryingMultiple.js>

In this example, the `add()` function takes three arguments `a`, `b`, and `c`. By invoking `add(5)`, we create a new function `addToFive` that expects the remaining two arguments.

Then, by invoking `addToFive(2)`, we create another new function `addToFiveAndTwo` that expects the final remaining argument. Finally, when we call `addToFiveAndTwo(3)`, it adds 5, 2, and 3 together, resulting in the output 10.

Function currying can be helpful in scenarios where you want to create reusable functions with partially applied arguments. It provides flexibility and allows you to build functions that are specific to different use cases while minimizing code duplication.

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

Repo Link –

* 1. 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.

Promises are an asynchronous programming concept in JavaScript that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. They provide a way to handle asynchronous operations in a more structured and manageable manner, avoiding callback hell and improving code readability.

A promise can be in one of three states:

**Pending:** This is the initial state of a promise. It means that the asynchronous operation associated with the promise is still in progress and hasn't been fulfilled or rejected yet.

**Fulfilled:** If the asynchronous operation is successful, the promise transitions to the fulfilled state. It means that the operation has completed successfully, and the promise holds the resulting value.

**Rejected:** If an error or failure occurs during the asynchronous operation, the promise transitions to the rejected state. It means that the operation was not successful, and the promise holds the reason or error for the failure.

To create your own promise, you can use the `Promise` constructor. It takes a callback function with two parameters: `resolve` and `reject`. Inside this callback function, you perform your asynchronous operation and call `resolve(value)` when it is successful or `reject(reason)` when it fails.

Here's an example where you create a promise to simulate a simple asynchronous delay:

Repo Link – <https://github.com/Gaurav-Dev24/Placement-Assignment/blob/main/JS%20Answers/Q%207/promise.js>

In this example, the `delay()` function returns a new promise. It uses the `setTimeout()` function to simulate an asynchronous delay. If the delay time (`ms`) is a non-negative value, the promise is resolved with a success message. Otherwise, the promise is rejected with an error message.

When we invoke `delay(2000)`, it returns a promise, to which we attach `.then()` and `.catch()` methods. The `.then()` method handles the fulfillment case by logging the success message, and the `.catch()` method handles the rejection case by logging the error message.

The output of the example would be:

```

Before delay

After delay

Delayed for 2000 milliseconds

```

**This demonstrates the asynchronous nature of promises. The delay of 2000 milliseconds doesn't block the execution of the code, allowing the "After delay" message to be logged before the promise is fulfilled.**

Promises provide a more elegant and manageable way to handle asynchronous operations in JavaScript, allowing you to chain multiple asynchronous operations, handle errors, and simplify the overall control flow of asynchronous code.

* 1. What is ‘this’ keyword in JavaScript? Explain with an example & create.

In JavaScript, the `this` keyword refers to the current execution context or the object that a function is bound to. It allows you to access properties and methods within the current context or object. The value of `this` depends on how a function is invoked.

Here are a few scenarios that determine the value of `this`:

**Global Scope:** When `this` is used in the global scope (outside of any function), it refers to the global object, which is `window` in a browser environment or `global` in Node.js.

```javascript

console.log(this); // Output: Window (in a browser environment)

```

**Function Invocation:** When `this` is used inside a regular function (not an arrow function), its value depends on how the function is invoked.

- In a regular function invocation, `this` refers to the global object (`window` in a browser) or `undefined` in strict mode.

```javascript

function greet() {

console.log(this);

}

greet(); // Output: Window (in a browser environment)

```

- If a function is part of an object and invoked using dot notation, `this` refers to the object that contains the function.

Repo Link – <https://github.com/Gaurav-Dev24/Placement-Assignment/blob/main/JS%20Answers/Q%208/thisFunctionInvocation.js>

**Method Invocation:** When a function is invoked as a method of an object, `this` refers to the object that the method belongs to.

Repo Link – <https://github.com/Gaurav-Dev24/Placement-Assignment/blob/main/JS%20Answers/Q%208/thisMethodInvocation.js>

**Constructor Invocation:** When a function is used as a constructor using the `new` keyword, `this` refers to the newly created object.

Repo Link – <https://github.com/Gaurav-Dev24/Placement-Assignment/blob/main/JS%20Answers/Q%208/thisConstructorInvocation.js>

**Explicit Binding:** You can explicitly bind `this` to a specific object using methods like `call()`, `apply()`, or `bind()`. These methods allow you to set the value of `this` explicitly when invoking a function.

Repo Link – <https://github.com/Gaurav-Dev24/Placement-Assignment/blob/main/JS%20Answers/Q%208/thisExplicitBinding.js>

In this example, `call()` is used to invoke the `greet` function and explicitly set `this` to the `person` object.

Understanding the `this` keyword is crucial for correctly accessing the current execution context or the object a function belongs to. Its value can vary depending on how a function is invoked, and it allows for dynamic and flexible behaviour in JavaScript.

* 1. Explain event loop, call Stack, callback, queue and micro task queue in your words.

Let me explain these concepts in simple terms:

**Call Stack:** The call stack is a data structure in JavaScript that keeps track of function calls. When a function is called, it gets added to the top of the call stack, and when a function finishes executing, it is removed from the stack. This allows JavaScript to keep track of which function is currently being executed.

**Event Loop:** The event loop is responsible for managing the execution of code in JavaScript. It continuously checks the call stack and other queues for tasks that need to be executed. Its primary job is to ensure that JavaScript remains single-threaded and handles asynchronous operations effectively.

**Callback Queue:** The callback queue, also known as the **task queue or message queue**, is a queue that holds callbacks or tasks that are ready to be executed. When an asynchronous operation, such as a setTimeout or an AJAX request, is completed, its callback function is placed in the callback queue.

**Micro Task Queue:** The micro task queue is a special queue that holds micro tasks. Micro tasks are functions that need to be executed asynchronously but with higher priority than regular tasks/callbacks. Promises and certain APIs, such as `queueMicrotask` and `Promise.resolve`, add tasks to the micro task queue.

The event loop follows a specific order when handling tasks:

- The call stack is checked. If it's empty, the event loop looks for tasks in the micro task queue.

- The micro task queue is emptied, and all tasks are executed in order until the queue is empty.

- If there are no micro tasks, the event loop looks for tasks in the callback queue.

- A single task/callback is taken from the callback queue and added to the call stack for execution.

- The call stack processes the task/callback, and if there are nested functions, they are added to the stack as well.

- Once the call stack is empty, the event loop starts over by checking the micro task queue.

In summary, the event loop coordinates the execution of code in JavaScript by managing the call stack, callback queue, and micro task queue. It ensures that tasks are executed in the correct order and that JavaScript remains responsive even when dealing with asynchronous operations.

* 1. Explain Debouncing and create a project where you are using debouncing.

Debouncing is a technique used in web development to optimize performance and prevent excessive or unnecessary function invocations, particularly in response to events like scrolling, resizing, or typing. It involves delaying the execution of a function until a certain period of inactivity has passed since the last invocation of that function.

The purpose of debouncing is to reduce the frequency of function calls, especially for events that can trigger rapidly in a short period. It helps to improve efficiency by ensuring that expensive or resource-intensive operations are only performed when necessary.

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

Repo Link – <https://github.com/Gaurav-Dev24/Placement-Assignment/blob/main/JS%20Answers/Q%2010/debouncing.html>

In this example, we have a search input field (`search-input`) and a div (`search-results`) to display the search results. The `debounce` function takes in a function (`fetchSearchResults`) and a delay time in milliseconds (300ms) and returns a new function that is debounced.

The `fetchSearchResults` function is the actual function that performs the search operation. It is wrapped within the `debounce` function, ensuring that it is invoked only after a 300ms delay of inactivity since the last input event.

The `fetchSearchResults` function simulates an API call by setting a timeout of 500ms. In a real scenario, this function would make an actual API call to fetch search results based on the input value.

The event listener is added to the search input field, and the debounced function (`debouncedSearch`) is invoked on the 'input' event. As the user types in the search input, the debounced function waits for a brief period of inactivity (300ms) before invoking the `fetchSearchResults` function.

By using debouncing, we ensure that the search operation is only triggered once the user has finished typing or after a short pause in typing, reducing unnecessary API calls and optimizing the performance of the search functionality.

In a production environment, it is recommended to use a well-tested and widely-used debounce implementation from a library like Lodash or Underscore.js.

* 1. Explain Closures and Use cases of Closures.

In JavaScript, a closure is a combination of a function and the lexical environment within which that function was declared. It allows a function to access variables from its outer scope, even after the outer function has finished executing. In simpler terms, a closure "closes over" the variables it needs, preserving their values and scope.

Closures are created when an inner function is returned or passed as a reference to an outer function. The inner function retains access to the variables, parameters, and even the outer function itself, even when the outer function has completed execution.

Here's an example to illustrate closures:

Repo Link – <https://github.com/Gaurav-Dev24/Placement-Assignment/blob/main/JS%20Answers/Q%2011/closure.js>

In this example, the `outerFunction` defines a variable `message` and an inner function `innerFunction`. The `innerFunction` references the `message` variable from its outer scope, forming a closure. When `outerFunction` is invoked and its inner function is returned, the returned function (assigned to `closure`) still has access to the `message` variable and can log its value.

Now, let's explore some use cases of closures:

**Data Privacy:** Closures are commonly used to create private variables and encapsulate data within a function. The variables within the outer function are inaccessible from the outside, but the inner function can still access and manipulate them. This allows you to control the visibility and accessibility of certain variables.

**Function Factories:** Closures are useful for creating function factories, where you can generate multiple functions with different preset values or configurations. The outer function acts as a factory that generates specific instances of the inner function with their own closed-over variables.

**Asynchronous Operations:** Closures are often employed in asynchronous operations, such as event handlers or callbacks. They allow you to maintain access to relevant data and state even after the initial function has finished executing.

**Memoization and Caching:** Closures can be used for memoization and caching purposes. By caching expensive computations or storing previously computed results within a closure, you can avoid redundant calculations and improve performance.

**Partial Application and Currying:** Closures play a significant role in techniques like partial application and currying, where you can create new functions by pre-filling some arguments. The closed-over variables capture the partially applied values, allowing for more flexible and reusable functions.

Closures provide a powerful and flexible way to manage data, encapsulate functionality, and create reusable functions with preserved state. They offer solutions to various programming problems and enable more advanced patterns and techniques in JavaScript.

* 1. Create a Blog web app using JavaScript

- Fetch data from <https://jsonplaceholder.typicode.com/posts>

- User can also add new blog

- Add Delete functionality also

Repo Link – <https://github.com/Gaurav-Dev24/Placement-Assignment/tree/main/JS%20Answers/Q%2012/Blog-App>

**React Answers**

* 1. What’s React and what are the advantages of it?

React is a popular JavaScript library for building user interfaces. It was developed by Facebook and is widely used for building interactive and scalable web applications. React follows a component-based architecture, where the user interface is divided into reusable components that manage their own state and can be composed together to create complex UIs.

Advantages of React include:

**Component-Based Development:** React promotes a modular approach to building UIs through reusable components.

**Virtual DOM:** React uses a virtual DOM (Document Object Model) to efficiently update and render UI elements. Instead of directly manipulating the actual DOM, React builds a virtual representation of it in memory and performs diffing to determine the minimal set of changes needed to update the UI. This approach results in faster rendering and improved performance.

**Unidirectional Data Flow:** React follows a unidirectional data flow, also known as one-way binding. Data flows from parent components to child components, making it easier to track and debug data changes. This approach enhances code predictability and maintainability.

**Declarative Syntax:** React utilizes a declarative syntax, where developers describe how the UI should look based on the current state of the application.

**Efficient Updates:** React efficiently updates only the necessary components and DOM elements when the underlying data changes.

**React Native:** React offers React Native, a framework for building native mobile applications.

**Large Community and Ecosystem:** React has a vast and active community of developers. This leads to extensive community-driven resources, libraries, tools, and support.

These advantages have contributed to the widespread adoption of React in the web development community. React empowers developers to create scalable, performant, and maintainable user interfaces, making it a popular choice for building modern web applications.

* 1. What's Virtual Dom in react & what are the advantages of it?

In React, the Virtual DOM (Document Object Model) is a lightweight, in-memory representation of the actual DOM. It is a concept and technique used by react to efficiently update and render user interfaces.

Advantages of the Virtual DOM in react include:

**Performance Optimization**: By using the Virtual DOM and its diffing algorithm, react minimizes the number of updates to the actual DOM. It calculates the most efficient way to update the UI based on the changes in the Virtual DOM, leading to improved rendering performance.

**Efficient Batch Updates**: React batches multiple state changes and props updates together during the diffing process. It then applies all the changes in a single update to the actual DOM. This reduces the number of expensive DOM operations and enhances performance.

**Developer-Friendly Abstraction:** The Virtual DOM provides a developer-friendly abstraction over the actual DOM. Developers can work with JavaScript objects representing the UI hierarchy instead of directly manipulating the DOM. This declarative approach simplifies UI development, makes the code more readable, and reduces the risk of introducing bugs.

**Cross-Platform Rendering:** The Virtual DOM enables React to support server-side rendering (SSR) and isomorphic/universal applications. With the ability to render components on both the server and the client, React provides better performance, SEO optimization, and a smoother user experience.

**Framework and Library Agnostic:** While the Virtual DOM is closely associated with react, its concept can be applied to other frameworks and libraries. Several other libraries have implemented their versions of the Virtual DOM, allowing for similar performance benefits in different ecosystems.

The Virtual DOM is a key feature of React that contributes to its efficiency, performance, and ease of development. It abstracts the complexity of the actual DOM and enables React to efficiently update and render UI components, resulting in a smoother user experience and optimized rendering performance.

* 1. Explain life cycle of react components?

In React, components have a lifecycle consisting of various stages and methods that are invoked at different points during the component's existence. These lifecycle methods allow you to perform specific actions at specific times, such as setting up initial state, handling updates, and cleaning up resources. The lifecycle of a react component can be divided into three main phases: mounting, updating, and un-mounting.

**Mounting Phase:**

- **constructor():** This is the first method called when a component is created. It is used to initialize state and bind event handlers.

- **staticgetDerivedStateFromProps():** This method is invoked before rendering and allows the component to update its internal state based on changes in props.

- **render():** This method returns the JSX that defines the component's UI structure. It should be a pure function that does not modify state or interact with the DOM.

- **componentDidMount():** This method is called immediately after the component is mounted in the DOM. It is used for side effects, such as making API calls or setting up subscriptions. DOM manipulation can also be performed here.

**Updating Phase:**

- **staticgetDerivedStateFromProps():** This method is called when the component is about to re-render due to changes in props. It allows the component to update its state based on the new props.

- **shouldComponentUpdate():** This method is called before re-rendering and determines whether the component should update. By default, it returns true, but you can implement custom logic to optimize rendering.

- **render():** The render method is called again to re-render the component with updated props and state.

- **componentDidUpdate():** This method is invoked after the component has been re-rendered. It is useful for performing side effects, such as updating the DOM based on the new state or props.

**Un-mounting Phase:**

- **componentWillUnmount():** This method is called right before the component is un-mounted from the DOM. It is used to clean up resources, such as cancelling timers or subscriptions.

It's important to note that some lifecycle methods are deprecated or will be removed in future React versions. The React team encourages the use of function components and hooks, which provide a more concise and flexible approach to managing component lifecycles.

* 1. What is the difference between Functional Components and Class Components?

Functional Components and Class Components are two types of components in React, each with its own syntax and capabilities. Here are the main differences between them:

**Syntax:**

- **Functional Components:** Functional Components are written as JavaScript functions. They receive props as an argument and return JSX elements that represent the UI.

- **Class Components:** Class Components are defined as JavaScript classes that extend the `React.Component` class. They use the **`render()`** method to return JSX elements.

**State Management:**

- **Functional Components:** Until React 16.8, functional components were stateless and could not manage state. However, with the introduction of React Hooks, functional components can now manage state using the **`useState`** hook and other hooks.

- **Class Components**: Class Components have built-in state management. They can define and update state using the **`this.state`** object and the **`setState()`** method.

**Lifecycle Methods:**

- **Functional Components:** Until React 16.7, functional components did not have lifecycle methods. However, with the introduction of React Hooks, functional components can now use lifecycle-related hooks such as **`useEffect`** to perform side effects and handle component lifecycle events.

- **Class Components:** Class Components have a variety of lifecycle methods, such as **`componentDidMount()`, `componentDidUpdate()`,** and **`componentWillUnmount()`,** which allow developers to perform actions at different stages of the component's lifecycle.

**Performance:**

- **Functional Components:** Functional Components are generally considered more lightweight and performant. They don't have the overhead of creating an instance of a class and have simpler rendering logic.

- **Class Components:** Class Components have a slightly higher overhead due to the creation of an instance and the additional lifecycle methods. However, the performance difference is often negligible in most applications.

**Code Organization:**

- **Functional Components:** Functional Components promote a more functional programming style, where the logic and behaviour of the component can be broken down into smaller functions. This can lead to a more modular and easier-to-understand code structure.

- **Class Components:** Class Components tend to contain all the logic and lifecycle methods within a single class, which can make the code structure less modular and harder to navigate in larger applications.

**React Hooks Support:**

- **Functional Components:** Functional Components fully support React Hooks, which allow you to use state and other React features without writing a class.

- **Class Components:** Class Components do not support React Hooks directly. To use Hooks, you would need to convert the class component into a functional component.

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

Hooks are a feature introduced in React 16.8 that allows functional components to use state and other React features without writing a class. They provide a way to reuse stateful logic and manage component lifecycle in functional components.

There are several built-in hooks available in React:

**useState:** useState hook allows functional components to have state. It returns an array with two elements: the current state value and a function to update the state value.

**useEffect:** useEffect hook is used to perform side effects in functional components. It is similar to **componentDidMount, componentDidUpdate, and componentWillUnmount** lifecycle methods in class components.

**useContext:** useContext hook enables functional components to consume values from a React context. It allows accessing the context value without the need for a context consumer component.

**useReducer:** useReducer hook is an alternative to useState that provides more control over complex state logic. It is similar to how Redux manages state using reducers.

**useCallback:** useCallback hook is used to memoize and optimize the creation of functions in functional components. It prevents unnecessary re-creation of functions on every render.

**useMemo:** useMemo hook allows memoization of expensive calculations in functional components. It only recomputes the value when the dependencies change.

**useRef:** useRef hook provides a way to create mutable references that persist across component renders. It can be used to store values or reference DOM elements.

**useLayoutEffect:** useLayoutEffect is similar to useEffect, but it runs synchronously after all DOM mutations. It is useful for performing DOM measurements and other imperative operations that need to be done before the browser paints.

Hooks are primarily designed to be used in functional components. However, since React 16.8, it is not possible to use hooks directly in class components. Hooks are only valid in the top-level of functional components or other custom hooks. If you have a class component and want to use hooks, you can consider converting the class component to a functional component to take advantage of hooks.

1. What are the Life cycle method and the advantages of it?

In React, lifecycle methods are special methods that are invoked at different stages of a component's existence. They allow you to perform specific actions at specific times, such as initializing state, handling updates, and cleaning up resources. Lifecycle methods provide hooks into the component's lifecycle events, giving you control over its behavior and allowing you to respond to changes in state and props.

Here are the main lifecycle methods in React class components:

**constructor():** The constructor method is called when a component is first initialized. It is used to set up the initial state, bind event handlers, and perform other setup tasks. It is the first method called in the lifecycle.

**render():** The render method is responsible for returning the JSX that defines the component's UI structure. It is a pure function that should not modify state or interact with the DOM directly.

**componentDidMount():** This method is invoked immediately after the component is mounted in the DOM. It is commonly used for performing side effects, such as making API calls, setting up subscriptions, or initializing third-party libraries. DOM manipulation can also be performed here.

**shouldComponentUpdate():** This method is called before the component is re-rendered. It determines whether the component should update or not based on the changes in props or state. By default, it returns true, but you can implement custom logic to optimize rendering and prevent unnecessary updates.

**componentDidUpdate():** This method is called after the component has been re-rendered due to changes in props or state. It is used for performing side effects that depend on the updated state or props. It is not called on the initial render.

**componentWillUnmount():** This method is invoked right before the component is unmounted and removed from the DOM. It is used to clean up resources, such as canceling timers, closing connections, or unsubscribing from subscriptions.

Advantages of lifecycle methods:

**Control and Flexibility**

**Side Effects**

**Optimization**

**Integration with External Libraries**

**Code Organization**

It's worth noting that with the introduction of React Hooks, functional components provide an alternative approach to managing component lifecycle and performing side effects. Hooks such as **`useEffect`** can be used to achieve similar functionality as lifecycle methods, but in a more concise and flexible manner.

1. What’s useState Hook & Advantages of it?

The `useState` hook is a built-in hook in React that allows functional components to manage state. It provides a simple and concise way to add stateful logic to 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 the state.

Here's an example of how to use the `useState` hook:

```jsx

import React, { useState } from 'react';

function Counter() {

const [count, setCount] = useState(0);

const increment = () => {

setCount(count + 1);

};

return (

<div>

<p>Count: {count}</p>

<button onClick={increment}>Increment</button>

</div>

);

}

```

Advantages of the `useState` hook:

**Simplicity:** The `useState` hook simplifies the process of managing state in functional components.

**Functional Programming:** Functional components with the `useState` hook promote a more functional programming style.

**Multiple State Variables:** The `useState` hook allows you to use multiple instances of state within a single component.

**Efficient Updates:** When using the `setState` function returned by `useState`, React intelligently updates only the specific state variable that has changed.

**Support for Complex Data Structures:** The `useState` hook can handle complex data structures as state, including objects and arrays.

**Synchronous Updates:** This means that consecutive state updates within a single function or event handler will be combined into a single update, reducing re-renders and improving performance.

**Compatibility with React DevTools**: The `useState` hook is fully compatible with React DevTools, allowing you to inspect and debug the state changes of your functional components.

The `useState` hook is a powerful feature in React that simplifies state management in functional components. It provides a concise syntax, efficient updates, and compatibility with React's development tools, making it a preferred choice for managing component state in modern React applications.

1. Explain useEffect & Advantages of it.

The `useEffect` hook is a built-in hook in React that allows functional components to perform side effects and handle component lifecycle events. It replaces the functionality of lifecycle methods like `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` in class components. The `useEffect` hook takes a callback function as its first argument and an optional array of dependencies as its second argument.

Here's an example of how to use the `useEffect` hook:

```jsx

import React, { useState, useEffect } from 'react';

function Timer() {

const [seconds, setSeconds] = useState(0);

useEffect(() => {

const timer = setInterval(() => {

setSeconds((prevSeconds) => prevSeconds + 1);

}, 1000);

return () => {

clearInterval(timer);

};

}, []);

return <div>Seconds: {seconds}</div>;

}

```

Advantages of the `useEffect` hook:

**Side Effects:** The `useEffect` hook allows you to handle side effects in functional components. Side effects can include making API calls, subscribing to events, manipulating the DOM, or interacting with external libraries. By placing the side effect logic within the `useEffect` callback, you ensure that it is executed at the appropriate times.

**Lifecycle Management:** The `useEffect` hook handles component lifecycle events automatically. The callback function specified in `useEffect` is executed after the component has rendered and the DOM has been updated.

**Dependency Management:** By providing an array of dependencies as the second argument to `useEffect`, you can control when the effect should run. The effect will only be re-executed if any of the dependencies have changed.

**Async Effects:** The `useEffect` hook supports handling async effects by using async/await or returning a promise from the effect callback.

**Multiple Effects:** You can use the `useEffect` hook multiple times within a single component to handle different side effects independently.

**Cleaner Code:** It eliminates the need for class components and lifecycle methods, resulting in cleaner and more concise code.

**Compatibility with React DevTools:** The `useEffect` hook is fully compatible with React DevTools, allowing you to inspect and debug the effects and dependencies of your functional components.

1. Explain Context Api and create a minor project on it.

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

Repo Link – <https://github.com/Gaurav-Dev24/Placement->Assignment/tree/main/React%20Answers/Dark-Light-theme/dar-light-mode

1. Explain useReducer and Its advantages.

The `useReducer` hook is a built-in hook in React that provides an alternative way to manage complex state and state transitions in functional components. It is inspired by the concept of reducers in Redux and allows you to manage state using a reducer function. The `useReducer` hook takes in a reducer function and an initial state, and returns the current state and a dispatch function to trigger state transitions.

Here's an example of how to use the `useReducer` hook:

```jsx

import React, { useReducer } from 'react';

// Reducer function

const reducer = (state, action) => {

switch (action.type) {

case 'INCREMENT':

return { count: state.count + 1 };

case 'DECREMENT':

return { count: state.count - 1 };

default:

return state;

}

};

function Counter() {

const [state, dispatch] = useReducer(reducer, { count: 0 });

const increment = () => {

dispatch({ type: 'INCREMENT' });

};

const decrement = () => {

dispatch({ type: 'DECREMENT' });

};

return (

<div>

<p>Count: {state.count}</p>

<button onClick={increment}>Increment</button>

<button onClick={decrement}>Decrement</button>

</div>

);

}

```

Advantages of the `useReducer` hook:

**State Management for Complex Scenarios:** The `useReducer` hook is suitable for managing complex state scenarios where state transitions are not straightforward. It allows you to encapsulate the state logic and transitions in a reducer function, providing a clear and predictable way to manage state.

**Centralized State Logic:** By using a reducer function, you can centralize state transitions and logic in a single place.

**Predictable State Updates:** The `useReducer` hook follows the principles of immutability. It ensures that state updates are done in an immutable manner, producing predictable and reliable state updates.

**Complex Actions and Payloads:** It gives you the flexibility to define custom action types and pass additional data as needed, making it suitable for scenarios that require more than simple state changes.

**Optimized Re-renders:** The `useReducer` hook, similar to `useState`, intelligently updates only the specific state variable that has changed. This optimization reduces unnecessary re-renders and improves performance.

**Testability:** The use of reducers with `useReducer` makes it easier to write unit tests for state transitions.

**Compatibility with React DevTools:** The `useReducer` hook is fully compatible with React DevTools, allowing you to inspect and debug state changes and actions in your functional components.

1. Build a Todo Web App Using React and useReducer Hook**.**

Repo Link –<https://github.com/Gaurav-Dev24/Placement-Assignment/tree/main/React%20Answers/Todo%20web%20app/todo>

1. Build a simple counter app using React.

Repo Link –

1. Build Calculator Using React Only.

Repo Link –<https://github.com/Gaurav-Dev24/Placement-Assignment/tree/main/React%20Answers/Calculator%20App/calculator>

1. Build a Tic Tac Toe Game using Class Component of React.

Repo Link –

1. Explain Prop Drilling & How can we avoid it?

Prop drilling refers to the process of passing props from a component to its child components through intermediary components that do not need the props themselves. It can occur when multiple levels of nested components need access to the same data or functions that are provided by a parent component.

Here's an example to illustrate prop drilling:

```jsx

// Parent component

function Parent() {

const data = "Hello, Prop Drilling!";

return <ChildA data={data} />;

}

// ChildA component

function ChildA({ data }) {

return <ChildB data={data} />;

}

// ChildB component

function ChildB({ data }) {

return <ChildC data={data} />;

}

// ChildC component

function ChildC({ data }) {

return <p>{data}</p>;

}

```

In this example, the `data` prop is passed down from the `Parent` component to `ChildA`, then to `ChildB`, and finally to `ChildC`. However, `ChildA` and `ChildB` do not actually use the `data` prop themselves; they only pass it down to their child components. This creates unnecessary coupling between components and can make the code harder to maintain and reason about.

To avoid prop drilling, there are a few approaches you can consider:

**Context API:** The Context API in React allows you to share data and functions across multiple levels of components without the need for explicit prop passing. It provides a way to create a global state that can be accessed by any component within the context. By using context, you can eliminate prop drilling and directly access the shared data or functions from any component that needs it.

**Redux or other State Management Libraries:** State management libraries like Redux provide a centralized store to manage state. With Redux, you can store data at a global level and access it from any component without the need for prop drilling. This approach is particularly useful for large-scale applications with complex state management needs.

**Component Composition:** Instead of passing props through intermediary components, you can compose components in a way that allows them to access the necessary data directly.

**Render Props or Function as Children:** Render props and the "function as children" pattern involve passing a function as a prop to a component, which that component can call to pass data back up the component hierarchy.

**Hooks:** With the introduction of hooks in React, such as `useContext` and `useReducer`, you can create custom hooks to encapsulate shared state or functionality and easily access it from any component without prop drilling.

By using one or a combination of these approaches, you can avoid prop drilling and create a more maintainable and scalable codebase in your React applications.

1. Create a task manager where user can create tasks and see his task

- Redirect him to task dashboard section after login

- Use https://reqres.in/ api to authenticate user and redirect him to task manager dashboard where he can see his task and create.

Repo Link –