**ASSIGNMENT**

**HTML Answers**

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

* HTML <! DOCTYPE> tag is used to inform the browser about the version of HTML used in the document. It is called as the document type declaration.
* The DOCTYPE declaration is an instruction to the web browser about what version of HTML the page is written in.
* It is used to define the type or version of html being used in webpage
* This ensures that the web page is parsed the same way by different web browsers.

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

* The semantic HTML tags help the search engines and other user devices to determine the importance and context of web pages.
* The pages made with semantic elements are much easier to read.
* It has greater accessibility. It offers a better user experience.
* Semantic HTML refers to syntax that makes the HTML more comprehensible by better defining the different sections and layout of web pages.
* Examples of semantic elements: <form> , <table> , and <article> - Clearly defines its content.
* The pages made with semantic elements are much easier to read. It has greater accessibility. It offers a better user experience.

1. Differentiate between HTML Tags and Elements?

* HTML Tags
* HTML tags are used to hold the HTML element.
* HTML tag starts with < and ends with >
* HTML tags are almost like keywords where every single tag has unique meaning.
* Ex- **<p> ………. </p>** is an example of tag. **<p>** is the known as opening tag and **</p>** is known as closing tag.
* HTML Elements
* HTML element holds the content.
* Whatever written within a HTML tag are HTML elements.
* HTML elements specifies the general content.
* For example: <p>This is sample content</p> represents HTML element.

1. Build Your Resume using HTML only.
2. **Write Html code so that it looks like the given image.**
3. What are some of the advantages of HTML5 over its previous versions?

* HTML5 introduced several significant advancements over its predecessors (HTML4 and XHTML) that enhanced web development capabilities.
* Here are some of the advantages of HTML5 are:-
* It is mobile-friendly and easy to use.
* Web pages contain a wide range of colors, shades, and different font types.
* HTML5 is compatible with all web browsers.
* Short and simple syntax.
* Improved security features.

1. Create a simple Music player using html only.
2. What is the difference between <figure> tag and <img> tag?

* <figure> Tag - The [<figure> tag](https://www.geeksforgeeks.org/html5-figure-tag/) in HTML5 is used to add self-contained content like illustrations, diagrams, photos, or codes listing in a document. It is related to the main flow, but it can be used in any position of a document and the figure goes with the flow of the document and if remove it then it should not affect the flow of the document.

Syntax for figure tag –

<figure>

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

<figcaption>Caption</figcaption>

</figure>

* <img>Tag - HTML [**<img>** tag](https://www.geeksforgeeks.org/html-img-tag/#:~:text=HTML%20tag%20is%20used,holds%20space%20for%20the%20image.&text=Attributes%3A%20The%20tag,the%20path%20to%20the%20image.) is used to add image or to set the background in the webpage/website. Nowadays website does not directly add images to a web page, as the images are linked to web pages by using the <img> tag which holds space for the image.

Syntax for figure tag –

<img src="url" alt="some\_text">

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

HTML Tag - HTML tags are used to define the structure and elements of an HTML document. They are enclosed in angle brackets (<>) and come in pairs—an opening tag and a closing tag. The opening tag denotes the start of an element, and the closing tag signifies the end.

For example:

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

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

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

HTML Attribute - HTML attributes provide additional information about an element. They are placed within the opening tag of an element and consist of a name-value pair. Attributes modify the behavior or appearance of an element, such as its color, size, alignment, or linking to external resources.

For example:

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">

Here are some examples of global attributes:

* Class - Specifies one or more CSS classes to associate with an element.
* Id - Provides a unique identifier for an element.
* Style – Defines inline CSS styles for an element, allowing direct control over its appearance.
* Title – Specifies a tooltip or additional information about an element, which is displayed when the user hovers over it.
* **Data -** Allows the addition of custom data attributes to elements.

1. Build Table which looks like the given image.

**CSS Answers**

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

The CSS box model is a container that contains multiple properties including borders, margin, padding, and the content itself. It is used to create the design and layout of web pages. According to the CSS box model, the web browser supplies each element as a square prism.

The properties associated with box model are:

* **Content** - The content of the box, where text and images appear.
* **Padding** - Clears an area around the content. The padding is transparent.
* **Border** - A border that goes around the padding and content.
* **Margin** - Clears an area outside the border. The margin is transparent.

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:-

1. **Element Selector** – Selects elements based on their tag name.

For example, **p** selects all **<p>** elements. The advantage of using element selectors is that they are simple and apply to all matching elements.

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

1. **Class Selectors -** The class selector selects HTML elements with a specific class attribute. It is used with a period character .

The advantage of class selectors is that they allow you to target specific groups of elements and apply styles consistently.

1. **ID Selectors** – The id selector selects the id attribute of an HTML element to select a specific element. An id is always unique within the page so it is chosen to select a single, unique element.

For example, **#my-id** selects the element with **id="my-id"**.

The advantage of ID selectors is that they provide a highly specific way to target individual elements.

1. **Attribute Selectors** – Select elements based on their attribute values. It uses square brackets (**[]**) and can match attributes in various ways.

For example, **[type="submit"]** selects elements with **type="submit"**.

The advantage of attribute selectors is that they allow you to target elements based on specific attribute conditions.

1. **Descendent Selectors** – Selects elements that are descendants of another element. It uses a space between two selectors.

For example, **div p** selects all **<p>** elements that are descendants of **<div>** elements.

The advantage of descendant selectors is that they provide a way to target specific elements within a hierarchical structure.

1. **Child Selectors -** Selects elements that are direct children of another element.

It uses the > symbol between two selectors.

For example, ul > li selects all <li> elements that are direct children of <ul> elements.

The advantage of child selectors is that they allow you to target specific elements at a specific level of the hierarchy.

1. **Pseudo-classes -** Selects elements based on a specific state or condition. Pseudo-classes start with a colon (:) and can be applied to various elements. For example, a:hover selects <a> elements when they are being hovered over. The advantage of pseudo-class selectors is that they provide dynamic styling based on user interactions or element states.
2. **Pseudo-elements Selectors -** Selects a specific part of an element. Pseudo-elements also start with a colon (:) and allow you to style specific parts of elements, such as the first line or first letter. For example, p::first-line selects the first line of <p> elements. The advantage of pseudo-element selectors is that they allow you to apply styles to specific parts of elements without modifying the HTML structure**.**
3. What is VW/VH & How it’s different from PX?

VW (viewport width) and VH (viewport height) are relative units of measurement in CSS that are based on the size of the viewport, which is the visible area of the web page.

VW Unit:1 VW represents 1% of the viewport width.

For example, if the viewport width is 1000 pixels, 1 VW is equal to 10 pixels (1% of 1000).

VH Unit:

1VH represents 1% of the viewport height.

Fexample:- if the viewport height is 800 pixels, 1 VH is equal to 8 pixels (1% of 800).

PX Unit:

PX (pixels) is an absolute unit of measurement in CSS.

It represents a fixed-size pixel on the screen.

For example, if you set an element's width to 200 pixels, it will always be 200 pixels wide regardless of the viewport size.

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 Block:-** Inline elements do not start on a new line. They flow along with the surrounding content horizontally.

They only take up as much width as necessary to accommodate their content.

Inline elements do not respect height, width, top/bottom margins, or padding properties.

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

**Inline Block:-** Inline-block elements share some characteristics with inline elements, but they also behave like block-level elements in some ways.

Inline-block elements flow along with the surrounding content horizontally, just like inline elements.

Examples of inline-block elements include <input>, <button>, and <select>.

**Block: -** Block-level elements start on a new line and take up the full width available, pushing subsequent elements onto a new line.

They can have a defined width, height, top/bottom margins, and padding.

Block-level elements respect height, width, top/bottom margins, and padding properties.

Examples of block-level elements include <div>, <p>, <h1> to <h6>, <ul>, and <li>.

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

**Content Box**

* The default value of the box-sizing property.
* The width and height of an element only include its content area.
* Padding and border are added to the total width and height of the element.
* When you set the width or height of an element, you are adjusting the dimensions of the content area, and the padding and border are added to it.
* Any additional width or height you specify for padding and border will increase the overall dimensions of the element.

**Border Box**

* The box-sizing: border-box; value alters the box model behavior.
* The width and height of an element include its content area, padding, and border.
* The specified width or height applies to the entire box, from the outer edge of the border to the opposite outer edge.
* The padding and border are included within the specified width or height, and they do not add to the overall dimensions of the element.

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 Flexbox are both CSS layout systems used to create responsive and flexible web page layouts.

**Flex box :**

Flexbox, or Flexible Box Layout, is primarily designed for one-dimensional layouts, either as a row or a column. It provides an efficient way to align, distribute, and reorder elements within a container.

**Key points about Flex box :** one dimensional layout, flex containers& flex items,

flexibility & alignment.

**Grid:**

CSS Grid Layout, commonly known as Grid, is a two-dimensional layout system

that allows for more complex and grid-based designs. It enables precise control over both rows and columns, allowing elements to be placed in specific grid cells or areas. Grid provides a powerful way to create responsive layouts with both flexible and fixed-sized elements.

**Key points about Grid box :** 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 the position values: absolute, relative, sticky, and fixed, along with examples for each.

**Absolute Positioning:**

* When an element is set to "position: absolute", it is positioned relative to its nearest positioned ancestor (an ancestor with a position value other than static) or the initial containing block if there is no positioned ancestor.

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:**

* With "position: relative", an element is positioned relative to its normal position in the document flow. It still occupies space in the normal flow, but its position can be adjusted using offset properties like top, right, bottom, and left.

Here's an example:-

**<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 hybrid of relative and fixed positioning. An element with "position: sticky" is initially positioned according to the normal flow of the document. However, as the user scrolls, it becomes "stuck" to a specific position within its containing element.

Here's an example:

**<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:**

* When an element has "position: fixed", it is positioned relative to the viewport, regardless of scrolling. The element remains fixed in its position even if the page is scrolled.

Here's an example:

<style>

.fixed {

position: fixed;

top: 50px;

right: 50px;

}

</style>

<div class="fixed">Fixed Position</div>

In this example, the fixed div is positioned 50 pixels from the top and 50 pixels from the right of the viewport. It will remain fixed at that position even if the user scrolls the page.

1. Build Periodic Table as shown in the below image.
2. Build given layout using grid or flex see below image for reference.
3. Build Responsive Layout both desktop and mobile and Tablet, see below image for reference?
4. What are Pseudo class in CSS & how it’s different From Pseudo Elements?

**JavaScript Answers**

1. **What is Hoisting in Javascript ?**

Hoisting is a JavaScript behavior where variable and function declarations are moved to the top of their containing scope during the compilation phase, before the code is executed. This means that regardless of where variables and functions are declared in the code, they are conceptually moved to the top of their scope.

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

JavaScript, higher-order functions are functions that can take other functions as arguments or return functions as results. They provide a powerful way to work with functions and data in a more functional programming style. Here are some common higher-order functions in JavaScript.

difference between .map()and .forEach():-

**. map() :-** The map() function is used to iterate over an array and transform each element of the array based on a provided callback function. It returns a new array with the transformed values, without modifying the original array.

**.forEach() :-** The forEach() function is used to iterate over an array and perform a provided callback function on each element of the array. It does not return a new array; instead, it is primarily used for executing side effects or performing actions on the elements

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

**.call() :-** The call() method is used to invoke a function with a specified this value and individual arguments passed directly as comma-separated values.

Here's an example:

**const person = {**

**name: "John Doe",**

**greet: function (greeting) {**

**console.log(greeting + ", " + this.name);**

**},**

**};**

**const otherPerson = {**

**name: "Jane Smith",**

**};**

**person.greet.call(otherPerson, "Hello");**

this example, the call() method is used to invoke the greet() function from the person object with the otherPerson object as the value of this. The additional argument "Hello" is passed as an argument to the function. This will output "Hello, Jane Smith".

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

Here's an example:

**const person = {**

**name: "John Doe",**

**greet: function (greeting, city) {**

**console.log(greeting + ", " + this.name + " from " + city);**

**},**

**};**

**const otherPerson = {**

**name: "Jane Smith",**

**};**

person.greet.apply(otherPerson, ["Hello", "New York"]);

this example, apply() is used to invoke the greet() function with the otherPerson object as the this value. The arguments "Hello" and "New York" are passed as an array. This will output "Hello, Jane Smith from New York".

**.bind() :-** The bind() method creates a new function with a specified this value and any initial arguments. It returns a new function without immediately invoking it.

Here's an example:

**const person = {**

**name: "John Doe",**

**greet: function (greeting) {**

**console.log(greeting + ", " + this.name);**

**},**

**};**

**const otherPerson = {**

**name: "Jane Smith",**

**};**

**const greetFunction = person.greet.bind(otherPerson);**

**greetFunction("Hello");**

this example, bind() is used to create a new function greetFunction with the otherPerson object as the value of this. The greetFunction can be invoked later, as shown. This will output "Hello, Jane Smith".

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

**Event bubbling :-** In event bubbling, when an event is triggered on an element, the event is first handled by the innermost element and then propagates upwards through its parent elements up to the outermost element (the document). This means that the event handlers of the parent elements will also be triggered unless the event propagation is explicitly stopped.

**Event Capturing:-** In event capturing, the event is handled by the outermost element (the document) first, and then it propagates downwards through its child elements until it reaches the innermost element. Event capturing is less commonly used than event bubbling but can be useful in certain scenarios.

1. **What is function currying with example?**

Function currying is a technique in functional programming where a function with multiple arguments is transformed into a sequence of functions, each taking one argument at a time. It allows you to create specialized versions of the original function by partially applying arguments.

Example:-

function multiply(a, b) {

return a \* b;

}

const multiplyByTwo = multiply.bind(null, 2); // Currying

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

this example, we have a multiply function that takes two arguments a and b and returns their product. We can use the bind() method to create a curried version of the multiply function by partially applying the first argument 2. The bind() method is used to set the this value (which is not relevant in this case) and the first argument of the function. The resulting function multiplyByTwo is a specialized version of multiply that multiplies a given number by 2.

When we invoke multiplyByTwo(4), it internally calls the original multiply function with the pre-set value of a as 2 and the argument b as 4. This results in a multiplication of 2 \* 4, which outputs 8.

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 a feature introduced in JavaScript to handle asynchronous operations more effectively and avoid the callback hell problem. A promise represents the eventual completion (or failure) of an asynchronous operation and provides a way to handle the result when it's available.

Promises have three different states:

* Pending: The initial state of a promise. It represents that the asynchronous operation is still in progress and the final result is not available yet.
* Fulfilled: The state of a promise when the asynchronous operation is successfully completed. It indicates that the promise has resolved with a value.
* Rejected: The state of a promise when the asynchronous operation encounters an error or fails. It indicates that the promise has been rejected with a reason or an error.

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.

**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.

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

Certainly! In JavaScript, the event loop, call stack, callback queue, and microtask queue are key components that work together to manage the execution of asynchronous code.

* Call Stack:

The call stack is a mechanism used by JavaScript to keep track of function calls. It works on a Last-In-First-Out (LIFO) principle. When a function is called, it is added to the top of the call stack, and when a function finishes executing, it is removed from the top.

* Event Loop:

The event loop is responsible for handling asynchronous operations in JavaScript. It constantly monitors the call stack and other queues to determine if there are any pending tasks.

* Callback Queue:

The callback queue (also known as the task queue or message queue) is a queue that holds callback functions waiting to be executed. When an asynchronous task, such as a timer or an event handler, is completed, its corresponding callback function is placed in the callback queue.

* Microtask Queue:

The microtask queue (also known as the job queue or promise queue) is a special queue that holds microtasks. Microtasks are a type of asynchronous tasks that have higher priority than regular tasks in the callback queue.

1. **Explain Debouncing and Create a project where you are using Debouncing.**
2. **Explain Closures and Use cases of Closures?**

Closures are an important concept in JavaScript that allow functions to retain access to variables from the outer (enclosing) scope, even after the outer function has finished executing. In simpler terms, a closure is a combination of a function and the lexical environment within which that function was declared.

cases of closures:-

* Data Privacy: Closures are often used to create private variables and encapsulate data. By defining variables within a closure, we can control their access and prevent direct modification from outside the closure.
* Function Factories: Closures can be used to create functions with preset configurations or parameters.
* Memoization: Closures can be used for memoization, which is a technique to cache the results of expensive function calls and avoid redundant computations.
* Event Handlers: Closures are commonly used in event handling scenarios. When attaching event listeners to DOM elements, closures can help in preserving the state of variables related to the event handler.

**React Answers**

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

React is a popular JavaScript library for building user interfaces. It is primarily used for creating reusable UI components and managing the state of those components efficiently. React follows a component-based architecture, allowing developers to create modular, reusable, and interactive UIs.

Advantage of React:-

Component-Based Approach: React promotes a modular and component-based approach to building UIs. Components encapsulate their own state and logic, making them reusable and easy to maintain.

Virtual DOM: React uses a virtual DOM (Document Object Model) to optimize rendering performance. It creates a lightweight in-memory representation of the actual DOM and performs a diffing algorithm to efficiently update only the necessary parts of the UI.

One-Way Data Binding: React enforces a unidirectional data flow, where data is passed from parent components to child components through props. This simplifies data management and makes it easier to track and debug changes in the application state.

JSX Syntax: React utilizes JSX (JavaScript XML), an extension to JavaScript, which allows developers to write HTML-like code directly in their JavaScript files. JSX makes the code more readable and intuitive, simplifying the process of creating and composing UI components.

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

The Virtual DOM (Document Object Model) is a concept in React that provides a lightweight in-memory representation of the actual browser DOM. It is a key aspect of React's efficient rendering and reconciliation process. The Virtual DOM acts as an intermediary layer between the application's state and the browser's DOM, facilitating faster updates and improved performance.

Advantages are:-

Performance Optimization: The Virtual DOM allows React to minimize the number of direct DOM manipulations, which are typically expensive operations.

Efficient Reconciliation: The diffing algorithm used by React to reconcile the Virtual DOM efficiently identifies the differences between the previous and new representations.

Abstraction from Browser-Specific APIs: The Virtual DOM provides a browser-agnostic abstraction layer. It enables React to work consistently across different browsers by providing a unified interface for manipulating the UI.

Simplicity and Productivity: The Virtual DOM simplifies the process of building complex UIs by abstracting away the complexities of directly interacting with the browser DOM.

1. **Explain LifeCycle of React Components?**

In React, the lifecycle of a component refers to the series of events or methods that are executed during the different stages of a component's existence. These lifecycle methods allow developers to perform specific actions at different points in the component's lifecycle, such as initialization, rendering, updating, and unmounting. The lifecycle methods are divided into three main phases: Mounting, Updating, and Unmounting.

1.Mounting Phase

2. Updating Phase

3. Unmounting Phase

1. **Whats the difference between 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.

**6. 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.

**7. 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.

**8. 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.

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**.**
2. Build a simple counter app using React.
3. Build Calculator Using React Only.
4. Build a Tic Tac Toe Game using Class Component of React.
5. 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.

**Express Question**

1. What is the difference between authentication and authorization?

Authentication and authorization are two related but distinct concepts in the field of computer security. Here's an explanation of the difference between the two:

**Authentication:-**

Authentication is the process of verifying the identity of a user or entity. It confirms whether the user is who they claim to be. In other words, authentication answers the question, "Are you who you say you are?" It ensures that a user is granted access to a system or resource only if they provide valid credentials, such as a username and password, a digital certificate, a biometric scan, or other authentication factors. The goal of authentication is to establish trust and ensure that only authorized individuals can access protected resources.

**Authorization:-**

Authorization, on the other hand, is the process of granting or denying access to specific resources or functionalities based on the authenticated user's privileges, rights, or permissions. It determines whether an authenticated user has the necessary privileges to perform certain actions or access certain resources within a system. Authorization is typically based on predefined rules, roles, or access control lists (ACLs) that define what actions or resources a user is allowed to access. It answers the question, "What are you allowed to do or access?"

Authentication is the process of verifying the identity of a user, while authorization determines what actions or resources a user is allowed to access based on their authenticated identity. Both authentication and authorization are important components of a secure system, working together to ensure that only authenticated and authorized users can access protected resources.

1. **What is he difference between common JS and EJS module?**

The difference between CommonJS and EJS modules lies in their origins, purpose, and syntax are:-

**CommonJS (CJS):-**

CommonJS is a module system used in older versions of JavaScript, primarily designed for server-side JavaScript environments like Node.js. It is based on synchronous loading and is mainly used for backend development. In CommonJS, modules are loaded using the require() function, and the module's exports are accessed using the module.exports object**.**

Here's an example of CommonJS syntax:

// Exporting module

module.exports = {

greet: function(name) {

console.log('Hello, ' + name + '!');

}

};

// Importing module

const myModule = require('./myModule');

myModule.greet('John');

**EJS Modules (ESM):-**

ESM (ECMAScript Modules) is the standard module system introduced in modern versions of JavaScript (ES6 and onwards). It is designed to work both in browsers and server-side environments. ESM supports asynchronous loading and has a more concise and expressive syntax. In ESM, modules are loaded using the import statement, and the module's exports are specified using the export keyword.

Here's an example of ESM syntax:-

javascript

Copy code

// Exporting module

export function greet(name) {

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

}

// Importing module

import { greet } from './myModule';

greet('John');

1. **What is JWT and what we can achieve with that create a minor project with jwt**
2. **What should we do with the password of a user before storing it into DB?**

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

Hashing:

Instead of storing the actual password, it should be hashed. Hashing is a one-way process that transforms the password into a fixed-length string of characters. A good cryptographic hash function, like bcrypt or Argon2, should be used for hashing. Hashing ensures that even if the database is compromised, the original passwords cannot be easily retrieved.

Salting:

To enhance password security, it is common to use salt along with hashing. A salt is a random value that is appended to the password before hashing. Each user should have a unique salt value. Salting ensures that even if two users have the same password, their hashed representations will be different, making it harder for attackers to use precomputed rainbow tables or dictionary attacks.

Iterations:

To slow down the hashing process and make it computationally expensive for attackers, multiple iterations of the hashing function should be performed. This helps protect against brute-force attacks and makes it harder for attackers to guess passwords by trying multiple combinations quickly.

Secure Storage:

Ensure that the database where the passwords are stored is properly secured. Implement measures such as access controls, encryption at rest, and regular security audits to protect against unauthorized access.

Protect Against Timing Attacks:

When comparing user-provided passwords during the authentication process, use a secure comparison method that avoids timing attacks. Timing attacks exploit the fact that string comparison may take a different amount of time based on how many characters match. Using a secure comparison function, like the one provided by your programming language or framework, helps prevent this vulnerability.

Educate Users About Strong Passwords:

Encourage users to create strong, unique passwords. Implement password policies that enforce complexity requirements (e.g., minimum length, including a mix of uppercase and lowercase letters, numbers, and special characters). Additionally, consider implementing measures like multi-factor authentication to add an extra layer of security.

1. **Whats event loop in NodeJS ?**

Node.js is a single-threaded event-driven platform that is capable of running non-blocking, asynchronous programming. These functionalities of Node.js make it memory efficient. The event loop allows Node.js to perform non-blocking I/O operations despite the fact that JavaScript is single-threaded. It is done by assigning operations to the operating system whenever and wherever possible.

Most operating systems are multi-threaded and hence can handle multiple operations executing in the background. When one of these operations is completed, the kernel tells Node.js, and the respective callback assigned to that operation is added to the event queue which will eventually be executed. This will be explained further in detail later in this topic.

**Features of Event Loop:**

* An event loop is an endless loop, which waits for tasks, executes them, and then sleeps until it receives more tasks.
* The event loop executes tasks from the event queue only when the call stack is empty i.e. there is no ongoing task.
* The event loop allows us to use callbacks and promises.
* The event loop executes the tasks starting from the oldest first.

**Phases of the Event loop:-**

* **timers:** this phase executes callbacks scheduled by setTimeout() and setInterval().
* **pending** callbacks: executes I/O callbacks deferred to the next loop iteration.
* **idle,** prepare: only used internally.
* **poll:** retrieve new I/O events; execute I/O related callbacks (almost all with the exception of close callbacks, the ones scheduled by timers, and setImmediate()); node will block here when appropriate.
* **check:** setImmediate() callbacks are invoked here.
* **close callbacks:** some close callbacks, e.g. socket.on('close', ...).

.