***Javascript Assignment 20***

***1). How does async/await help with performance and scalability?***

*Async/await is a programming construct in JavaScript that allows for asynchronous programming without using callbacks or promises explicitly. It is built on top of promises and enables developers to write asynchronous code that is more readable, maintainable, and easier to reason about.*

*Async/await can help with performance and scalability in several ways:*

*Non-blocking I/O: Async/await can help to avoid blocking the event loop, which is a common performance bottleneck in Node.js applications. By using asynchronous code, the event loop can continue to process other requests while the I/O operation is being performed, leading to better performance and scalability.*

*Parallelism: Async/await enables parallelism by allowing multiple asynchronous operations to be executed concurrently. This can be achieved by using the Promise.all() method, which waits for all the promises to resolve before returning the results.*

*Error handling: Async/await makes it easier to handle errors in asynchronous code by using try/catch blocks. This makes the code more maintainable and easier to debug, as error handling is centralized and explicit.*

*Code readability: Async/await makes it easier to write and understand asynchronous code by allowing developers to write code that looks like synchronous code. This can lead to more readable and maintainable code, which in turn can improve the scalability of the application.*

***2). Is it possible to use async/await with promise chains? If yes,***

***how can this be achieved?***

*Yes, it is possible to use async/await with promise chains. The basic idea is to wrap the promise chain in an async function and use await to handle the resolution of each promise in the chain. Here's an example:*

*async function myAsyncFunction() {*

*try {*

*const result1 = await myPromiseFunction1();*

*const result2 = await myPromiseFunction2(result1);*

*const result3 = await myPromiseFunction3(result2);*

*return result3;*

*} catch (error) {*

*console.error(error);*

*}*

*}*

*In this example, myPromiseFunction1, myPromiseFunction2, and myPromiseFunction3 are functions that return promises. The await keyword is used to wait for each promise to resolve before moving on to the next one in the chain.*

*Here, we are using a try/catch block to handle any errors that may occur in the promise chain. This is a best practice when using async/await with promise chains, as it allows us to handle errors in a more readable and understandable way.*

*We can use async/await with promise chains by wrapping the chain in an async function and using await to handle the resolution of each promise in the chain.*

***3). Give 3 real world examples where async/await has been used?***

***Web Applications:*** *async/await is commonly used in web development for handling asynchronous operations, such as making API calls or fetching data from a database. For example, when building a web application that requires data to be retrieved from a server, async/await can be used to make the HTTP requests and handle the responses.*

***Node.js Applications:*** *async/await is also used in building server-side applications with Node.js, such as web servers, chat applications, and streaming services. Node.js is built on an event-driven architecture and is well-suited for asynchronous operations, and async/await can make it easier to write and maintain asynchronous code.*

***Desktop Applications:*** *async/await can also be used in desktop application development. For example, in a video editing application, async/await can be used to process large video files in the background while the user continues to work on other tasks. Similarly, in a photo editing application, async/await can be used to apply filters or edits to images in the background while the user works on other parts of the application.*

***4. ANSWER:***

*The output of the code will be 3.*

***5. ANSWER:***

*The output of the code will be:*

*1*

*Error: some error*

***6. ANSWER:***

*The output of the code will be:*

*3*

*1*

*2*

*Go!*

***7. ANSWER:***

*The output of the code will be:*

*10*

***8). Is it possible to nest async functions in JavaScript? Explain with examples.***

*Yes, it is possible to nest async functions in JavaScript. An async function returns a promise, which can be consumed by another async function using the await keyword. Here's an example:*

*async function f1() {*

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

*setTimeout(() => {*

*console.log("f1 completed");*

*resolve(1);*

*}, 1000);*

*});*

*}*

*async function f2() {*

*console.log("f2 started");*

*let result = await f1();*

*console.log(`f1 returned ${result}`);*

*console.log("f2 completed");*

*return 2;*

*}*

*async function f3() {*

*console.log("f3 started");*

*let result = await f2();*

*console.log(`f2 returned ${result}`);*

*console.log("f3 completed");*

*}*

*f3();*

*In this example, we have three async functions - f1, f2, and f3. f1 simulates an asynchronous operation that takes 1 second to complete and returns the value 1. f2 calls f1 using the await keyword and logs the result to the console. f3 calls f2 using the await keyword and logs the result to the console.*

***9). What is the best way to avoid deadlocks when using***

***async/await?***

*Deadlocks occur when two or more asynchronous operations are waiting for each other to complete, causing the program to become stuck. To avoid deadlocks when using async/await, we should follow these best practices:*

1. *We should avoid using nested await calls, as this can lead to deadlocks. Instead, use Promise.all to run multiple asynchronous operations in parallel.*
2. *We should use timeouts or other methods to prevent long-running or infinite loops from blocking other asynchronous operations.*
3. *We should use error handling to catch and handle any exceptions or errors that may occur during the asynchronous operation. This can prevent the program from becoming stuck in an infinite loop or waiting for an event that may never occur.*

***10). In which scenarios would you use synchronous code instead of asynchronous code?***

*There are certain scenarios in which we might prefer to use synchronous code instead of asynchronous code. Here are a few examples:*

***When the operation is not I/O-bound:*** *If the operation does not involve any I/O, such as performing a simple calculation or accessing data from memory, then using synchronous code may be simpler and more efficient.*

***When the operation is time-sensitive:*** *If the operation needs to be completed within a very short time frame, then using synchronous code may be a better option, as it can ensure that the operation is completed as quickly as possible.*

***When the code is simple:*** *If the code is simple and does not require complex error handling or concurrency management, then using synchronous code may be simpler and more straightforward.*

***When the code is small:*** *If the code is small and does not involve complex logic, then using synchronous code may be easier to read and maintain than asynchronous code.*

***When the benefits of using asynchronous code do not outweigh the complexity:*** *While asynchronous code can provide many benefits, such as improved performance and scalability, it can also be more complex and difficult to maintain. If the benefits of using asynchronous code do not outweigh the additional complexity, then using synchronous code may be a better option.*

*Overall, the choice between using synchronous and asynchronous code depends on the specific requirements and constraints of the application, as well as the developer's expertise and experience.*