**Async/Await Assignment**

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

I. Increase the performance and responsiveness of your application, particularly when you have long-running operations that do not require to block the execution. In this case, you can perform other work while waiting for the result from the long running task.

II. Organize your code in a neat and readable way significantly better than boilerplate code of the traditional thread creation and handling. with async / await , you write less code and your code will be more maintainable than using the previous asynchronous programming methods such as using plain tasks.

III. async / await is the newer replacement to BackgroundWorker, which has been used on windows forms desktop applications.

IV. You make use of the latest upgrades of the language features, and there have been some improvements added to the feature like foreach async and generalized async type like ValueTask.

2). Is it possible to use async/await with promise chains? If yes, how can this be achieved?

 The await keyword pauses the execution of the async function and waits for the passed Promise 's resolution, and then resumes the async function's execution and returns the resolved value. The await keyword is only valid inside async functions.

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

I. Displaying the progress of an async method.

II. Pausing the progress of an async method.

III. Using Task.WhenAll() to wait for all tasks to complete.

4).

async function inc(x) {

x = x + await 1

return x;

}

async function increment(x){

x = x + 1

return x

}

inc(1).then(function(x){

increment(x).then(function(x){

console.log(x)

})

})

Find output.

PS C:\Users\ADMIN\Documents\c\JS\_CODE> node assign\_20.js

3

5).

let p = new Promise(function (resolve, reject) { reject(new Error("some error"));

setTimeout(function(){

reject(new Error("some error"));

},1000)

reject(new Error("some error")); });

p.then(null, function (err) { console.log(1);

console.log(err);

}).catch(function (err) {

console.log(2);

console.log(err);

});

Find output.

PS C:\Users\ADMIN\Documents\c\JS\_CODE> node assign\_20.js

1

Error: some error

6).

async function f1() {

console.log(1);

}

async function f1() {

console.log(2);

}

console.log(3);

f1();

console.log(1);

f2();

async function f2() {

console.log("Go!");

}

Find output.

PS C:\Users\ADMIN\Documents\c\JS\_CODE> node assign\_20.js

3

2

1

Go!

7).

function resolveAfterNSeconds(n,x) { return new Promise(resolve => {

setTimeout( ( ) => {

resolve(x);

}, n);

});

}

(function(){

let a = resolveAfterNSeconds(1000,1)

a.then(async function(x){

let y = await resolveAfterNSeconds(2000,2) let z = await resolveAfterNSeconds(1000,3) let p = resolveAfterNSeconds(2000,4)

let q = resolveAfterNSeconds(1000,5)

console.log(x+y+z+await p +await q);

})

})()

Find output.

PS C:\Users\ADMIN\Documents\c\JS\_CODE> node assign\_20.js

15

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

Yes,

An API that returns a [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) will result in a promise chain, and it splits the function into many parts. Consider the following code:

function getProcessedData(url) {

return downloadData(url) // returns a promise

.catch((e) => downloadFallbackData(url)) // returns a promise

.then((v) => processDataInWorker(v)); // returns a promise

}

it can be rewritten with a single async function as follows:

async function getProcessedData(url) {

let v;

try {

v = await downloadData(url);

} catch (e) {

v = await downloadFallbackData(url);

}

return processDataInWorker(v);

}

Alternatively, you can chain the promise with catch():

async function getProcessedData(url) {

const v = await downloadData(url).catch((e) => downloadFallbackData(url));

return processDataInWorker(v);

}

In the two rewritten versions, notice there is no await statement after the return keyword, although that would be valid too: The return value of an async function is implicitly wrapped in [Promise.resolve](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/resolve) - if it's not already a promise itself.

9). What is the best way to avoid deadlocks when using async/await?

 1. Use ConfigureAwait(ContinueOnCapturedContext : false) in all async library methods

2. Make the Api Controller Get() await the library method call and return a Task instead

Note : Async all the way

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

1. Sequential tasks, where stopping the entire program to wait for a network request or disk IO makes sense.

2. tasks are performed one at a time and only when one is completed, the following is unblocked.