Mohan Sai Potla

29 Oct 2022

JavaScript Assignment 20

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

Async/await helps to improve the performance and scalability by allowing application to do other work while it is waiting for a task to complete.

- ➔ It can make User Interface more responsive to the user.
- ➔ If we have more time taking I/O, async operations save much time.
- ➔ They make code execution more efficient by allowing tasks to be run in parallel

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 promises. Here is an example with fetch api which is usually a promise.

```
const fetchAndAddTodos = async () => {
    try {
        const response = await
fetch("https://jsonplaceholder.typicode.com/todos/1");
        const todo = await response.json();
        console.log( todo );
   }
   catch(error) {
       console.log( error.message );
   }
};

fetchAndAddTodos();
```

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

-> Async/await can be used in place of callbacks to avoid callback hell.

-> Async/await is nothing more than syntactic sugar for promises.

-> Async function provides us with a clean and concise syntax that enables us to write less code to accomplish the same outcome we would get with promises.

4) Find output.

```
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)
    })
})
```

//Output:

3

5). Find output.

```
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);
});
```

//Output:

1

Error: some error

6). Find output.

```
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!");
}
```

```
Output:
```

3

2

1

Go!

7). Find output.

```
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);
    })
})()
```

```
Output:
```

`15`

`Above code gives output 15 after 7 seconds.`

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

Yes. It is possible to next async functions in JavaScript.

```
async function foo() {

    await setTimeout(() => console.log("Inside foo"),3000)

    async function faa() {
        await setTimeout(() => console.log("Inside faa"),3000)
        console.log(2);
      }

    faa();
    console.log("faa called")
    console.log(1);
}
```

```
foo();
```

```
Output:
faa called
1
2
Inside foo
Inside faa
```

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

The best way to avoid deadlocks when using async/await is to async all the way down to prevent deadlocks. Then it will be all asynchronous not synchronous blocking. Main thing to remember is that we should not block on tasks.

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

In synchronous code, tasks are performed one at a time and only when one is completed, the other task will start.

In asynchronous code, we can move to another task before the previous one finishes.

We use asynchronous code for requesting data from a network, accessing a database and for handling time taking events. In remaining all tasks which consumes less time, we will use synchronous coding.