Asynchronous Javascript

Introduction

Asynchrony in javascript enables our code to be executed after a certain time interval. Unlike synchronous code, asynchronous code does not block the execution during the time it waits for execution. Asynchrony in javascript can be achieved through the following methods:

- Callbacks
- Promises
- setTimeout() and setInterval()

Callbacks

A callback is a function which is passed as an argument to another function which can be invoked later on by that function. It is used to complete a sequence of actions.

Example:

```
function doATask(callback) {
    console.log("Task 1 done!")
    callback()
}

function yetAnotherTask() {
    console.log("Task 2 done!")
}

doATask(yetAnotherTask)
```

```
// Output:
// Task 1 done!
// Task 2 done!
```

Callbacks are used to delay the execution of a task and to be executed after the completion of a certain task.

Problem With Callbacks

When writing large applications we have to deal with multiple asynchronous tasks that needs to be done one after the other and this causes a large amount of nested callbacks. This code becomes very unreadable and difficult to maintain which causes a **callback hell**. To escape this problem we use promises instead of callbacks to handle asynchronous tasks.

Promises

Promises are used to handle asynchronous tasks in JavaScript. Managing them is easier when dealing with multiple asynchronous operations, where we'll get stuck in callback hell, created by callbacks which ultimately leads to unmanageable code.

A Promise is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason.

A promise is always in one of these states:

- Pending
- Resolved
- Rejected

Creating Promises

We can create a promise using the promise constructor:

```
const promise = new Promise((resolve,reject) => {
    resolve(1)
})
```

Promise constructor takes a function as a single argument, which takes two arguments, **resolve** and **reject**. We will call resolve if everything inside the function goes well, else we call reject. A pending promise can be resolved by providing a value or rejected by providing a reason (error).
If any of these options occur, we must take the appropriate steps. The methods promise.then() and promise.catch() are used to take any further action with a promise that becomes settled.

then():

When a promise is resolved or rejected, then() is called. Two functions are passed to the then() method. If the promise is resolved and a result is received, the first function is called. If the promise is rejected and an error is returned, the second function is called. (It's optional as the catch() method is comparatively a better way to handle errors.)

Example:

```
promise.then((data) => {
    // On Resolved
}, (error) => {
    // On Rejected
})
```

catch():

catch() is called to handle the errors, i.e., when a promise is rejected or when some error has occurred during the execution. catch() method takes one function as an argument, which is used to handle errors.

Example:

```
promise.catch((error) => {
    // Handle Error on Rejected or caught Error
})
```

Passing Parameters to resolve/reject

You can also pass parameters to resolve and reject! For example, the above promise can also be written as:

Example:

```
const promise = new Promise((resolve,reject) => {
    const num = Math.floor(Math.random() * 100)
    if(num%2) resolve("odd")
    else reject("even")
})

promise.then((data) => {
    console.log(data)
}).catch((error) => {
    console.log(error)
})
```

Ideally, you should always wrap the promise within a function, which in turn will return the promise. This will increase the readability of your code and it'll also add meaning to it.

Example:

```
function oddEven(){
    return promise = new Promise((resolve,reject) => {
        const num = Math.floor(Math.random() * 100)
        if(num%2) resolve("odd")
        else reject("even")
    })
}

oddEven().then((data) => {
    console.log(data)
}).catch((error) => {
    console.log(error)
})
```

Chaining Multiple Handlers

Multiple callback functions would create a callback hell that leads to unmanageable code. So many function calls will lead to confusion and it'll become harder for us to understand. We can use promises in chained requests to avoid callback hell and easily avoid callback hell.

Example:

```
function oddEven(){
    return promise = new Promise((resolve,reject) => {
        resolve(1)
    })
}

oddEven().then((data) => {
    console.log(data) // 1
    return data*2
})
.then((data) => {
    console.log(data) // 2
    return data*3
})
.then((data) => {
    console.log(data) // 3
})
.catch((error) => {
    console.log(error)
})
```

Promise Methods

There are some methods defined inside of the Promise API which helps us deal with a bulk of promises at once:

- Promise.race()
- Promise.all()
- Promise.any()
- Promise.allSettled()

Promise.race():

The `Promise.race()` method takes an iterable of promises as an input, and returns a single Promise that resolves into the first resolved or rejected promise.

Example:

```
const promise1 = () => new Promise((resolve,reject) => {
    setTimeout(() => {
        resolve(1)
    },100)
})

const promise2 = () => new Promise((resolve,reject) => {
    resolve(2)
})

const promise3 = () => new Promise((resolve,reject) => {
    setTimeout(() => {
        resolve(3)
    })
})


Promise.race([promise1(),promise2(),promise3()])
.then((val) => {
    console.log(val) // 2
})
.catch((err) => {
    console.log(err)
})
```

Promise.all():

The `Promise.all()` method takes an iterable of promises as an input, and returns a single promise that resolves to an array of the results of the input promises. This returned promise will resolve when all of the input's promises have resolved, or if the input iterable contains no promises. It rejects immediately upon any of the input promises rejecting or non-promises throwing an error, and will reject with this first rejection message / error.

Example:

```
const promise1 = () => new Promise((resolve,reject) => {
    setTimeout(() => {
        resolve(1)
    },100)
})

const promise2 = () => new Promise((resolve,reject) => {
    resolve(2)
})

const promise3 = () => new Promise((resolve,reject) => {
    setTimeout(() => {
        resolve(3)
    })
})


Promise.all([promise1(),promise2(),promise3()])
.then((val) => {
    console.log(val) // [1,2,3]
})
.catch((err) => {
    console.log(err)
})
```

Promise.any():

`Promise.any()` takes an iterable of **Promise** objects and, as soon as one of the promises in the iterable fulfills, returns a single promise that resolves with the value from that promise. If no promises in the iterable fulfill (if all of the given promises are rejected), then the returned promise is rejected with an **AggregateError**, a new subclass of **Error** that groups together individual errors.

```
const promise1 = () => new Promise((resolve,reject) => {
    setTimeout(() => {
        resolve(1)
    },100)
})

const promise2 = () => new Promise((resolve,reject) => {
    reject(2)
})

const promise3 = () => new Promise((resolve,reject) => {
    setTimeout(() => {
        resolve(3)
    })
})


Promise.any([promise1(),promise2(),promise3()])
.then((val) => {
    console.log(val) // 3
})
.catch((err) => {
    console.log(err)
})
```

Promise.allSettled():

The **Promise.allSettled()** method returns a promise that resolves after all of the given promises have either fulfilled or rejected, with an array of objects that each describes the outcome of each promise.

```
const promise1 = () => new Promise((resolve,reject) => {
    setTimeout(() => {
        resolve(1)
    },100)
})

const promise2 = () => new Promise((resolve,reject) => {
    reject(2)
})

const promise3 = () => new Promise((resolve,reject) => {
    setTimeout(() => {
        resolve(3)
    })
})


Promise.allSettled([promise1(),promise2(),promise3()])
.then((val) => {
    console.log(val) // 3
})
.catch((err) => {
    console.log(err)
})

// Output:
// [
//       { status: 'fulfilled', value: 1 },
//       { status: 'rejected', reason: 2 },
//       { status: 'fulfilled', value: 3 }
// ]
```

Async/Await Syntax Sugaring

An async function is a function declared with the **async** keyword, and the **await** keyword is permitted within them. The **async** and **await** keywords enable asynchronous, promise-based behavior to be written in a cleaner style, avoiding the need to explicitly configure promise chains.

Example:

Consider the following code snippet:

```
const promise1 = () => new Promise((resolve,reject) => {
    setTimeout(() => {
        resolve("resolved 1!",100)
    })
})

const promise2 = () => new Promise((resolve,reject) => {
    setTimeout(() => {
        resolve("resolved 2!",100)
    })
})

function asyncAwait() {
    promise1()
    .then((result) => {
        console.log(result)
        return promise2()
    })
    .then((result) => {
        console.log(result)
    })
}

asyncAwait()
```

It can also be written like this using async/await:

```
const promise1 = () => new Promise((resolve,reject) => {
    setTimeout(() => {
        resolve("resolved 1!",100)
    })
})

const promise2 = () => new Promise((resolve,reject) => {
    setTimeout(() => {
        resolve("resolved 2!",100)
    })
})

async function asyncAwait() {
    const result1 = await promise1()
    const result2 = await promise2()

    console.log(result1)
    console.log(result2)
}

asyncAwait()
```

Await can only be used before those functions which returns a promise. Async functions always return a promise.

Error Handling in Async/Await

Await statements make asynchronous calls behave exactly like synchronous calls. So we can use try-catch statements to catch errors in await statements.

Example:

```
const promise1 = () => new Promise((resolve,reject) => {
    setTimeout(() => {
        resolve("resolved 1!",100)
    })
})

const promise2 = () => new Promise((resolve,reject) => {
    setTimeout(() => {
        reject("rejected 2!",100)
    })
})

async function asyncAwait() {
    try{
        const result1 = await promise1()
        const result2 = await promise2()

        console.log(result1)
        console.log(result2)
    } catch(e) {
        console.log(e) // rejected 2!
    }
}

asyncAwait()
```

This type of error handling doesn't work on synchronous functions calling asynchronous tasks.

Event Loop

Event loop is the key to the asynchronous nature of javascript. The main task of the event loop is to take callbacks from the task queue and microtask queue and put them in the call stack. This operation is performed only when the call stack is empty. If there are tasks present in both the queues then it gives priority to the microtask queue. It pulls out callbacks from the microtask queue until it is empty and then starts with the task queue. This is the reason promises have a higher priority than other asynchronous methods.

Task Queue:

All the callbacks from setTimeout() and setInterval() are pushed into this queue after the timer ends and then when the call stack is empty and microtask queue is cleared, the event loop will take the callbacks from here and push it to the call stack.

Microtask Queue:

All the callbacks from the promises are pushed into this queue and then when the call stack is empty, the event loop will take the callbacks from here and push it to the call stack.

You can read a detailed explanation on event loop here: The Node.js Event Loop, Timers, and process.nextTick() | Node.js (nodejs.org)