Promises Deep Dive

Promises:

In a simplest approach, we were using async and await function.

How does a promise work?

Promise is also an object which implies it will have some functions and properties on it. They are the states of the promise.

3-States:

- Pending – something that is scheduled as a future event at a particular time.
- Fulfilled – if the promise is fulfilled,
- Rejected – did no keep up the promise

Relate this to your day to day promise.

Await keyword is not stopping all the commands, but only the execution of the next line has been paused. It would waiting for that particular line, but the remaining events would be taking place at the same time and only once the promise is fulfilled will JS move forward.

Previously promise was creating for me, but how would we as a developer be creating a promise.

```
const myPromise = new Promise()

console.log(nmyPromise)
```

We need to create a parameter for it to work. We have to give one parameter, called the executor function. We will have to pass a function, directly inline or we can put in another function declared.

```
const myPromise = new Promise(function () {
    console.log('My Promise')
})

console.log(nmyPromise)
```

My promise shows status pending as we are not doing anything with it. Like .then(), the function that you pass through the constructor as an argument receives two parameters. By convention, the first parameter is called resolve and the second is called reject.

```
const myPromise = new Promise(function (resolve, reject) {
    console.log('My Promise')
})

console.log(nmyPromise)
```

The resolve and reject are again functions. If we console.log them, we can see the output as in ***f() {[native code]}.***

```
const myPromise = new Promise(function (resolve, reject) {
    console.log('My Promise')
})

console.log(myPromise)

myPromise.then(function () {
    console.log('Coming from then execution function')
})
```

There is a function which is being referred to the .then() function. If we call the resolve() function, the output would show the console.logged statement. When you create a promise, you need to pass in the first parameter as the resolve.

```
function xyz(){
    console.log('Coming from then execution function')
}
myPromise.then(xyz)
```

This resolve will be a function, which is the same function that is passed in to the .then() function. If we create another function xyz() and if we pass xyz in the .then() function, then the resolve will be actually referring to the function. In my promise resolve will actually be referring to the xyz() function.

For operations taking particular time, after the process is done, if that operation is successful, it will become fulfilled now so, whenever the promise is fulfilled, the argument passed would be fulfilled. It changes the state of the argument would be fulfilled. As the function xyz was passed, the resolve function would call in xyz because that is what we are passing to it.

Instead if we are doing some process and some error occurred, reject would be called then it would first update the state of promise to rejected and subsequently it will call the function .catch().

Now, call the abc() when the promise is rejected. So when I'm calling the promise in the executor function, it would change the state to reject and subsequently abc() would be called. This is how promises internally works.

```
const myPromise = new Promise(function (resolve,reject) {
    reject()
    resolve()
})

console.log(myPromise)

function xyz() {
    console.log('Coming from xyz function')
}

function abc() {}

myPromise.then(xyz)
myPromise.catch(abc)
```

setTimeOut():

```
const myPromise = new Promise(function (resolve,reject) {

    setTimeOut(function () {
        resolve()

    }, 5000)
    // reject()
    // resolve()
})

myPromise.then(xyz)
myPromise.catch(abc)

function xyz() {
    console.log('Coming as promise is fulfilled')
}

function abc() {
    console.log('Coming as promise is rejected')
}

console.log('This is logged immediately')
```

This is logged immediately is show as soon the page refreshes, but the resolve() function has been delayed for 5 sec. We can do a fetch request as well instead of doing this delay. This is a good example of asynchronous operation. Let’s put some logic in the setTimeOut function.

```
const myPromise = new Promise(function (resolve, reject) {

    setTimeOut(function () {
        const number = 12

        if (number % 2 === 0) {
            resolve()
        } else {
            reject()
        }

    }, 5000)
})

myPromise.then(xyz)
myPromise.catch(abc)

function xyz() {
    console.log('Coming as promise is fulfilled')
}

function abc() {
    console.log('Coming as promise is rejected')
}

console.log('This is logged immediately')
```

This entire function will called after 5 seconds and depending on the number, either resolve() or reject() function would be called.

When we fetch, it would return a promise on which a .then is given to be called. This is getting called only when the data is fetched. So internally fetch is also doing the same thing. Once the data is inside the chrome browser then fetch is calling my resolve() function. The moment any error occurs, catch() function would be executed and internally reject() function would be executed.

If we have some utility function written on top or at the end of the file. We want pass the const number to xyz(). We would do this bbny passing parameters to the resolve() itself.

```
function cube(){

}
```

We could actually pass the number into the resolve and xyz. Cube() is going to accept one parameter as argument. This number will not be reflecting the same variable.

```
function cube(){
     num1 * num1 * num1
}
```

If it’s a user function

```
const myPromise = new Promise(function (resolve,reject) {

    setTimeOut(function () {
        const number = 12

        if (number % 2 === 0) {
            const numberCube = cube(number)
            resolve(numberCube)
        } else {
            reject()
        }

    }, 5000)

})
myPromise.then(xyz)
myPromise.catch(abc)

function xyz(numberCube) {
    console.log('Coming as promise is fulfilled')
    console.log(numberCube)
}
```

Creating an Ajax Request:

```
const url = 'https://jsonplaceholder.typicode.com/todos'

const request = new XMLHttpRequest();
// open a connection to a URL
request.open('get', url)
```

In order to define what should happen if the request is successful, we need onreadystatechange. Every statechange should trigger the function that has been given.

```
request.onreadystatechange = () => {

}
```

The first parameter would be something called as the data itself. But we have to actually look for the state change. So,

```
request.onreadystatechange = (data) => {
    if (request.readyState == 4 && request.status == 200){

        const jsonData = JSON.parse()
    }
}
```

200 is basically a way of say that we are able to get the data. Once this is true then we can do whatever we want with the data. If the request is successful, there is property responseText which is going to be assigned.

```
const request = new XMLHttpRequest();
// open a connection to a URL
request.open('get', url)

request.onreadystatechange = (data) => {
    if (request.readyState == 4 && request.status == 200){

        const jsonData = JSON.parse(request.responseText)
          console.log(jsonData)
    }
}
```

Now, we need to call another function to send the data. This will actually send the data to the server and thereby calling an ajax call.