Day 18 - Promises

Promise

Humans give or receive promises to do some activity at some point in time, such as “I promise to take out the trash” or “you promised to wash the dishes”. If you keep your promise, you’ll make others happy, but if you don’t, it may lead to discontent. A promise in JavaScript has something in common with a regular human promise.

Promise is a way to handle asynchronous operations in JavaScript. It allows handlers with an asynchronous action’s eventual success value or failure reason. This allows asynchronous methods to return values like synchronous methods; instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.

Promise states:

- pending: initial state, neither fulfilled nor rejected.
- fulfilled: meaning that the operation completed successfully.
- rejected: meaning that the operation failed.

A pending promise can either be fulfilled with a value, or rejected with a reason (error). When either occurs, the associated handlers queued up by a promise’s then method are called. (If the promise has already been fulfilled or rejected when a corresponding handler is attached, the handler will be called, so there is no race condition between an asynchronous operation completing and its handlers being attached). As the Promise.prototype.then() and Promise.prototype.catch() return promises, they can be chained.

Callbacks

The following code blocks will show the difference between a callback and promises.

The example below will contain a callback function that takes two parameters. The first is err and the second is result. If the err parameter is false, there will not be an error. Otherwise, there will be an error.

```
// Callback
const doSomething = callback => {
  setTimeout(() => {
    const skills = ['HTML', 'CSS', 'JS']
    callback('It did not go well', skills)
  }, 2000)
}

const callback = (err, result) => {
  if (err) {
    return console.log(err)
  }
  return console.log(result)
}

doSomething(callback)

//log (after two seconds)
It did not go so well.
```

In this case, the err is false and will return the else block, which is the result

```
const doSomething = callback => {
  setTimeout(() => {
    const skills = ['HTML', 'CSS', 'JS']
    callback(false, skills)
  }, 2000)
}

doSomething((err, result) => {
  if (err) {
    return console.log(err)
  }
  return console.log(result)
})

//log (after two seconds)
['HTML', 'CSS', 'JS']
```

Promise Constructor

We create promises using the Promise constructor. We can create a new promise using the keyword *new* followed by the word Promise and followed by a parenthesis. In the parenthesis, it takes a callback function. The callback function has two parameters which are the resolve and reject functions.

```
// syntax
const promise = new Promise((resolve, reject) => {
  resolve('success')
  reject('failure')
})
```

```
// Promise
const doPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    const skills = ['HTML', 'CSS', 'JS']
    if (skills.length > 0) {
      resolve(skills)
    } else {
      reject('Something went wrong')
    }
  }, 2000)
})

doPromise
  .then(result => {
    console.log(result)
  })
  .catch(error => console.log(error))
```

The above promise is settled with resolve. Lets use another example when the promise is settled with reject.

```
const rejPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
        const skills = ['HTML', 'CSS', 'JS']
        if(skills.includes('Node')) {
            resolve('fullstack developer')
        } else {
            reject('Something went wrong')
        }
    }, 4000)
})

rejPromise
    .then(result => {
```

```
        console.log(result)
    })
    .catch(error => {
        console.log(error)
    })
```

Fetch API

The Fetch API provides an interface for fetching resources (including across the network). It'll seem familiar to anyone who has used XMLHttpRequest, but the API provides a more powerful and flexible feature set. Lets use fetch to request URL and API's. Additionally, we'll go over promise use cases in accessing network resources using the fetch API.

```
const url = 'https://restcountries.com/v2/all' // countries api
fetch(url)
    .then(response => response.json()) // accessing the API data as JSON
    .then(data => {
        for(let i = 0; i < 5; i++) {
            console.log(data[i].name) // getting the data
        }
    })
    .catch(error => console.error(error)) // handling error if something wrong happens
```

Async and Await

Async and await are elegant ways to handle promises. It's easy to understand and clean to write.

```
const square = async function (n) {
  return n * n
}
square(2)

//log:
Promise{<resolved>: 4}
```

Async in front of a function means that the function will return a promise. The above square function returns a promise instead of a value.

How can we access the value from the promise? To access, we use the keyword await.

```
const square = async function(n) {
    return n * n
}
const readSquare = async function(n) {
    const value = await square(n)
    console.log(value)
}
readSquare(3)
```

As seen above, writing async in front of a function creates a promise, and to get the value from a promise, we use await. Async and await go together, one can't exist without the other.

Below, we'll use the fetch API data using both promise method and async and await method.

- promise

```
const url = 'https://restcountries.com/v2/all'
fetch(url)
  .then(response => response.json())
  .then(data => {
    console.log(data)
  })
  .catch(error => console.error(error))
```

- async and await

```
const fetchData = async () => {
  try {
    const response = await fetch(url)
    const countries = await response.json()
    console.log(countries)
  } catch(err)
    console.error(err)
  }
}
console.log('===== async and await')
fetchData()
```

```
const fetchData = async () => {
  try {
    const response = await fetch(url)
    const countries = await response.json()
    console.log(countries)
  } catch(err)
    console.error(err)
  }
}
console.log('===== async and await')
fetchData()
```