Asynchronous JavaScript Promises Explained

🚀 Everything about JS Promises in just **20 pages: Part 1**

Arpitha Rajeev

arpitha.rajeev37@gmail.com

Agenda

- Promises
- How Promises solve Callback Hell?
- Using Promises
- Handling Errors in Promises
- Promise Terminology
- Response Object
- Chaining Promises
- Resolving Promises
- finally() Method
- catch() Method

Arpitha Rajeev
arpitha.rajeev37@gmail.com

Promises

- **Promises** is a core language feature designed to simplify asynchronous programming
- A **Promise** is an **object** that represents the **result** of an **asynchronous computation**
- **Asynchronous computation** will not have a result ready immediately
- As long as the result is not available, the state of the **Promise** is **pending** and its **result is undefined**

Arpitha Rajeev
arpitha.rajeev37@gmail.com

Promises

```
1  const user = fetch('https://jsonplaceholder.typicode.com/todos/1');
```

```
▼ user: Promise
  ▶ [[Prototype]]: Promise
    [[PromiseState]]: "pending"
    [[PromiseResult]]: undefined
```

- Using **fetch** we are calling an URL, **fetch** returns a **Promise Object** stored in **user**
- Before getting the result, **user** contains a **Promise Object** with its **state** as **pending** and **result** as **undefined**

Arpitha Rajeev
arpitha.rajeev37@gmail.com

Promises

```
▶ [[Prototype]]: Promise
  [[PromiseState]]: "fulfilled"
▶ [[PromiseResult]]: Response
```

- Once the response is ready, **user** will contain a **Promise Object** with its **state** as **fulfilled** and the **result** will contain the **Response Object**
- To get the value, we ask the Promise to call a **callback function** when the value is ready

Arpitha Rajeev
arpitha.rajeev37@gmail.com

How Promises solve Callback Hell

- **Promises** allow **nested callback** to be re-expressed as a more **linear Promise chain** that tends to be easier to read
- **Promises** invoke the callback function **only once** when the value is ready
- This resolves issues associated with **Inversion of Control**, the outer function might have invoked the callback function never or more than once

Arpitha Rajeev
arpitha.rajeev37@gmail.com

How Promises solve Callback Hell

- But **Promise** will guarantee to invoke the function only once when the value is ready
- For this reason, they can be used to replace **setTimeout**() but not **setInterval**() because the latter invokes the function multiple times
- In the case of a button click, we will not use **Promise** as we want to allow the user to click a button multiple times
- **Promise** also handle errors efficiently

Arpitha Rajeev
arpitha.rajeev37@gmail.com

Using Promises

- **getJSON**() function returns a **Promise Object** when called with an **url**
- Promise object defines a **then**() instance method that takes a callback function that will be invoked only once
- When a response is received, this callback function is invoked

```
getJSON(url).then(jsonData => {
// This is a callback function that will be asynchronously
// invoked with the parsed JSON value when it becomes availabl
});
```

Arpitha Rajeev
arpitha.rajeev37@gmail.com

Handling Errors in Promises

```
getJSON(url).then(success).catch(error);
```

- **getJSON**() function completes normally and returns a Promise Object but an error occurs in the callback function **success**
- **.then**() method also returns a Promise that has a **.catch**() instance method that will invoke a **callback function called error** in this case

Arpitha Rajeev
arpitha.rajeev37@gmail.com

Promise Terminology

- In real life, we use the terms Promise is **kept** or Promise is **broken**
- In JavaScript, Promise is **fulfilled** or Promise is **rejected** are equivalent to kept and broken
- Promise can never be both fulfilled and rejected, when we don't have an error it is **fulfilled**, when there is an error it is **rejected**
- Once a promise is fulfilled or rejected, we say that it is settled and its value can't be changed

Arpitha Rajeev
arpitha.rajeev37@gmail.com

Promise Terminology

- When it is **fulfilled**, the result of the Promise is the **return value** of the code
- When it is rejected, then the value is an error of some sort
- **Fulfilled:** async code runs normally
- **Rejected:** async code throws an error
- **Settled:** Promise is either fulfilled or rejected
- Promise can have only 3 states at a time: **Pending**, **Fulfilled** and **rejected**

Arpitha Rajeev
arpitha.rajeev37@gmail.com

Response Object

```
▼ Promise {<fulfilled>: Response} i
  ▶ [[Prototype]]: Promise
    [[PromiseState]]: "fulfilled"
  ▼ [[PromiseResult]]: Response
      body: (...)
      bodyUsed: false
    ▶ headers: Headers {}
      ok: true
      redirected: false
      status: 200
      statusText: ""
      type: "cors"
      url: "https://jsonplaceholder.typicode.com/todos/1"
    ▶ [[Prototype]]: Response
```

Arpitha Rajeev
arpitha.rajeev37@gmail.com

Chaining Promises

- To avoid **callback hell** (Pyramid of doom), we use promise chaining
- This way our code expands vertically instead of horizontally, chaining is done using **'.then**()'
- **Promise-based Fetch API** will return a Promise Object, when it is fulfilled it passes a **Response Object** to the function passed in **then**() method
- The **Response Object** as shown above will give access to the headers, statusText, status

Arpitha Rajeev
arpitha.rajeev37@gmail.com

Chaining Promises

```
fetch('https://jsonplaceholder.typicode.com/todos/1')
.then((item)=>{
    return item.json()
})
.then((result)=>{
    console.log(result)
    return result
})
.then((final)=>{
    console.log(final)
})
.catch((error)=>{
    console.log(error)
})
```

Arpitha Rajeev
arpitha.rajeev37@gmail.com

Chaining Promises

- When fetch returns a Promise object, **then**() is invoked on it that returns a new Promise object
- It is not fulfilled until the function passed to **then**() is complete
- On the first line, fetch() is invoked with a URL which is **task 1** of initiating a **HTTP GET request** and returns a Promise called **promise 1**
- On the second line, the then() method of promise 1 is invoked, when promise 1 is fulfilled it invokes callback function with **item** as **argument**

Arpitha Rajeev
arpitha.rajeev37@gmail.com

Chaining Promises

- First callback function returns a new **Promise** that is provided as input the second **then**() method that invokes callback when fulfilled
- The last **then**() method also returns a **Promise** but it is not used but its value is logged
- If there is an error at any stage, **catch**() **method** is called that is used to log the error
- Response Object has **.text**(), **.json**() and other methods that returns a **Promise Object**

Arpitha Rajeev
arpitha.rajeev37@gmail.com

Resolving Promises

- The first **then**() method invokes a callback function that returns another Promise object using **return item.json**()
- When a callback returns, **Promise p** is resolved with a **value v** and if this **value v** is not a Promise then we say **Promise p** is fulfilled immediately
- But if the **value v** is a Promise like in our example, then we say **Promise p** is **resolved** and **not fulfilled yet** which will be **settled** only when **value v** (which is a Promise) is **settled**

Arpitha Rajeev
arpitha.rajeev37@gmail.com

.finally() Method

- If we add a **.finally**() invocation to the Promise chain, then the callback passed to **.finally**() will be invoked when the Promise called it on settles
- **.finally**() **returns a new Promise object** whose return value is generally ignored
- The Promise returned by **.finally**() will typically **resolve** or **reject** with the same value that the Promise that .finally() was invoked on resolves or rejects with

Arpitha Rajeev
arpitha.rajeev37@gmail.com

.catch() Method

- The callback passed to **.catch**() will only be invoked if the callback at a previous stage throws an error
- If the callback returns **normally**, then the **.catch**() callback will be **skipped**
- The return value of the previous callback will become the input to the next .then() callback
- **.catch**() callbacks are not just for reporting errors, but for handling and recovering from errors

Arpitha Rajeev
arpitha.rajeev37@gmail.com

.catch() Method

```
startAsyncOperation()
.then(doStageTwo)
.catch(recoverFromStageTwoError)
.then(doStageThree)
.then(doStageFour)
.catch(logStageThreeAndFourErrors);
```

- If either **startAsyncOperation**() or **doStageTwo**() throws an error, then **recoverFromStageTwoError**() function will be invoked

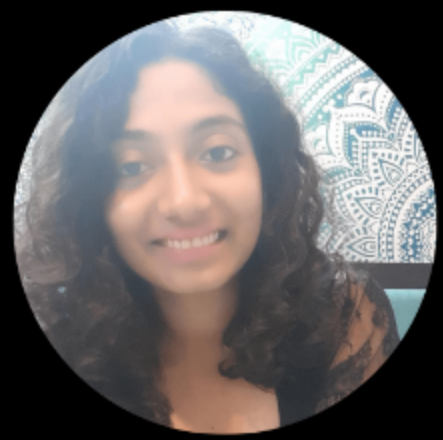
Arpitha Rajeev
arpitha.rajeev37@gmail.com

.catch() Method

- If **recoverFromStageTwoError**() returns normally, its return value will be passed to **doStageThree**()
- The asynchronous operation continues normally
- If **recoverFromStageTwoError**() was unable to recover, it will itself throw an error
- In this case, neither **doStageThree**() nor **doStageFour**() will be invoked
- Error thrown by **recoverFromStageTwoError**() would be passed to **logStageThreeAndFourErrors**()

Arpitha Rajeev
arpitha.rajeev37@gmail.com

Follow Me



For more such content on Software Development

Arpitha Rajeev

arpitha.rajeev37@gmail.com