Episode 2 : Promises

> Promises are used to handle async operations in JavaScript.

We will discuss with code example that how things used to work before `Promises` and then how it works after `Promises`

Suppose, taking an example of E-Commerce

```
const cart = ['shoes', 'pants', 'kurta'];

// Below two functions are asynchronous and dependent on each other
const orderId = createOrder(cart);
proceedToPayment(orderId);

// with Callback (Before Promise)
// Below here, it is the responsibility of createOrder function to first create
the order then call the callback function
createOrder(cart, function () {
  proceedToPayment(orderId);
});
// Above there is the issue of `Inversion of Control`
```

Q: How to fix the above issue?
A: Using Promise.

Now, we will make `createOrder` function return a promise and we will capture that `promise` into a `variable`

Promise is nothing but we can assume it to be empty object with some data value in it, and this data value will hold whatever this `createOrder` function will return.

Since `createOrder` function is an async function and we don't know how much time will it take to finish execution.

So the moment `createOrder` will get executed, it will return you a `undefined` value. Let's say after 5 secs execution finished so now `orderId` is ready so, it will fill the `undefined` value with the `orderId`.

In short, When `createOrder` get executed, it immediately returns a `promise object` with `undefined` value. then javascript will continue to execute with other lines of code. After sometime when `createOrder` has finished execution and `orderId` is ready then that will `automatically` be assigned to our returned `promise` which was earlier `undefined`.

Q: Question is how we will get to know `response` is ready?
A: So, we will attach a `callback` function to the `promise object` using `then` to get triggered automatically when `result` is ready.

```
const cart = ['shoes', 'pants', 'kurta'];

const promiseRef = createOrder(cart);
// this promiseRef has access to `then`

// {data: undefined}
// Initially it will be undefined so below code won't trigger
// After some time, when execution has finished and promiseRef has the data then
automatically the below line will get triggered.

promiseRef.then(function () {
  proceedToPayment(orderId);
});
```

Q: How it is better than callback approach?

In Earlier solution we used to pass the function and then used to trust the function to execute the callback.

But with promise, we are attaching a callback function to a promiseObject.

There is difference between these words, passing a function and attaching a function.

Promise guarantee, it will callback the attached function once it has the fulfilled data. And it will call it only once. Just once.

Earlier we talked about promise are object with empty data but that's not entirely true, `Promise` are much more than that.

Now let's understand and see a real promise object.

fetch is a web-api which is utilized to make api call and it returns a promise.

We will be calling public github api to fetch data https://api.github.com/users/alok722

```
// We will be calling public github api to fetch data
const URL = 'https://api.github.com/users/alok722';
const user = fetch(URL);
// User above will be a promise.
console.log(user); // Promise {<Pending>}

/** OBSERVATIONS:
 * If we will deep dive and see, this `promise` object has 3 things
 * `prototype`, `promiseState` & `promiseResult`
 * & this `promiseResult` is the same data which we talked earlier as data
 * & initially `promiseResult` is `undefined`
 *
 * `promiseResult` will store data returned from API call
 * `promiseState` will tell in which state the promise is currently, initially it
will be in `pending` state and later it will become `fulfilled`
 */

/**
 * When above line is executed, `fetch` makes API call and return a `promise`
instantly which is in `Pending` state and Javascript doesn't wait to get it
`fulfilled`
 * And in next line it console out the `pending promise`.
 * NOTE: chrome browser has some in-consistency, the moment console happens it
shows in pending state but if you will expand that it will show fulfilled because
chrome updated the log when promise get fulfilled.
 * Once fulfilled data is there in promiseResult and it is inside body in
ReadableStream format and there is a way to extract data.
 */
```

Now we can attach callback to above response?

Using `.then`

```
const URL = 'https://api.github.com/users/alok722';
const user = fetch(URL);

user.then(function (data) {
  console.log(data);
});
// And this is how Promise is used.
// It guarantees that it could be resolved only once, either it could be `success`
or `failure`
/**
    A Promise is in one of these states:

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

```
    rejected: meaning that the operation failed.
 */
```

💡 Promise Object are immutable.
-> Once promise is fulfilled and we have data we can pass here and there and we don't have to worry that someone can mutate that data. So over above we can't directly mutate `user` promise object, we will have to use `.then`

Interview Guide

💡 What is Promise?
-> Promise object is a placeholder for certain period of time until we receive value from asynchronous operation.

-> A container for a future value.

-> **A Promise is an object representing the eventual completion or failure of an asynchronous operation.**

We are now done solving one issue of callback i.e. Inversion of Control

But there is one more issue, callback hell...

```
// Callback Hell Example
createOrder(cart, function (orderId) {
  proceedToPayment(orderId, function (paymentInf) {
    showOrderSummary(paymentInf, function (balance) {
      updateWalletBalance(balance);
    });
  });
});
// And now above code is expanding horizontally and this is called pyramid of
doom.
// Callback hell is ugly and hard to maintain.

// 💡 Promise fixes this issue too using `Promise Chaining`
// Example Below is a Promise Chaining
createOrder(cart)
  .then(function (orderId) {
    proceedToPayment(orderId);
  })
  .then(function (paymentInf) {
    showOrderSummary(paymentInf);
  })
  .then(function (balance) {
```

```
    updateWalletBalance(balance);
  });

// ⚠️ Common PitFall
// We forget to return promise in Promise Chaining
// The idea is promise/data returned from one .then become data for next .then
// So,
createOrder(cart)
  .then(function (orderId) {
    return proceedToPayment(orderId);
  })
  .then(function (paymentInf) {
    return showOrderSummary(paymentInf);
  })
  .then(function (balance) {
    return updateWalletBalance(balance);
  });

// To improve readability you can use arrow function instead of regular function
```

Watch Live On Youtube below:

Edit this page