Promises

Tuesday, 22 August 2023 1:47 PM

→ Promises are used to handle asynchronous operations in JS.

eg: Let's consider we are creating an E-commerce website.
It has 2 APIs: createOrder() and proceedToPayment().
Now, proceedToPayment() must run after createOrder() is run.
createOrder() returns orderId which we pass to proceedToPayment().

→ Before Promises, we did this, to achieve above functionality:

```
1  const cart = ["shoes", "pants", "kurta"];
2
3  createOrder(cart, function (orderId) {
4    proceedToPayment(orderId);
5  });
6
```

→ we used callback fn's previously & thus,

all our control of code went to createOrder API & it controls whether proceedToPayment() will be executed or not.

→ Now, we can achieve above using Promises:

```
6
7  const promise = createOrder(cart);
8
9  // {data: undefined }
```

→ Let's assume that createOrder() API returns us a promise. Therefore it will give us an empty object with data value in it & it will hold whatever createOrder() API returns to us.

→ Now createOrder() API is an async operation, it will take some time to execute. As soon as Line 7 is executed it will return us an object with an object having data with undefined value as reflected in line 9.
Thus, as soon as line 7 is executed it will return a promise which is an empty object having data with undefined value, now the program will not wait for the object to be filled with values and will instead go on executing the next lines of code.
After some time the promise object will be filled with data automatically. And we will get order details in it after whatever async time it takes.

```
6
7  const promise = createOrder(cart);
8
9  // {data: orderDetails }
```

→ Now, to continue with the program, we attach a callback fn to this

promise object. In order to do this we use .then().
.then() is a fn that is available over promise object.
Now, whenever the promise object gets filled with value or data that is returned by createOrder() API, the callback fn that we have attached to the promise will run automatically.

```
7   const promise = createOrder(cart);
8
9   // {data: orderDetails }
10
11  promise.then(function (orderId) {
12    proceedToPayment(orderId);
13  });
14
```

→ How are Promises better than callbacks?

→ In callbacks we are passing a fn to another fn but using promises we are attaching the callback fn. So previously createOrder() had all the control for when proceedToPayment() will be run but now, we have the control for the same.

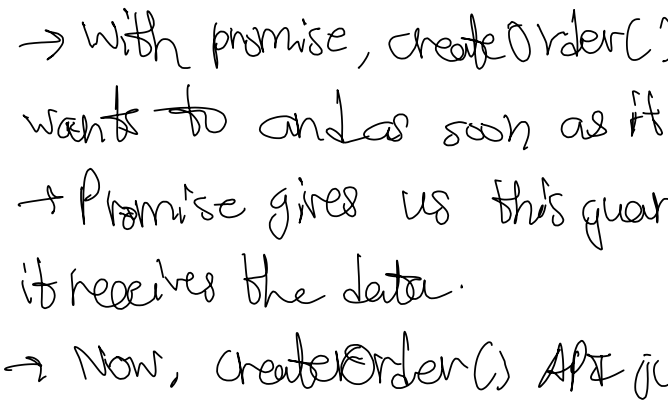
→ With promise, createOrder() API will fill promise variable whenever it wants to and as soon as it fills the data, our callback fn will be called.

→ Promise gives us this guarantee that it will run the callback fn as soon as it receives the data.

→ Now, createOrder() API just needs to do its own job & the control of when the callback fn will be called is with us. So this solves the problem of inversion of control.

→ Also, JS gives us the guarantee that callback fn would be called only once when using promise.

→ eg. of promise using fetch():



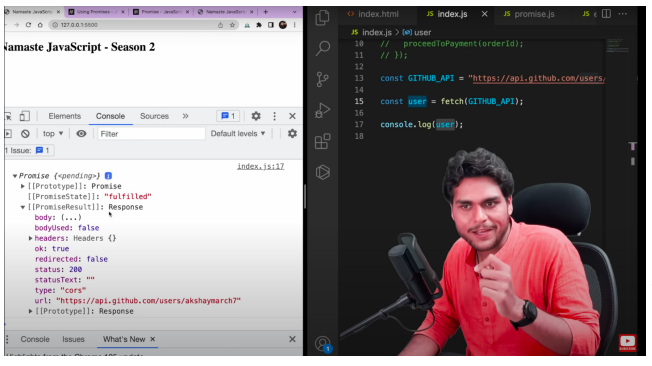
→ Here we've used an open source API and passed the URL to fetch().

→ We can see in the browser that we get a promise with pending state.

→ Result stores whatever data the fetch() will return from the server.

→ state tells you in what state that promise is in. Initially it will be in pending state. When we get the data, pending state of promise changes to fulfilled state.

→ Kind of inconsistency in Chrome console:



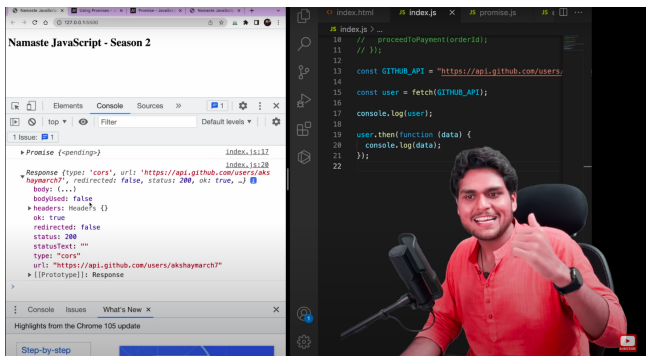
→ When we console, the promise is in pending state.

→ In line 15, when fetch() returns a promise, at that point of time promise is in pending state.

→ It takes some time for it to get fulfilled but JS does not wait for that, it just goes & prints the promise in the console.

→ So, some time later, data will actually come inside result in promise so, Chrome shows state as fulfilled in console. So we can assume that at this point of time, promise has been fulfilled. But during the time it was consoled, it was in pending state.

→ Now, we can attach a callback fn to this promise & do whatever we want to do with data.



→ Promise can also have a rejected state.

Q: What are the 3 states of a promise?

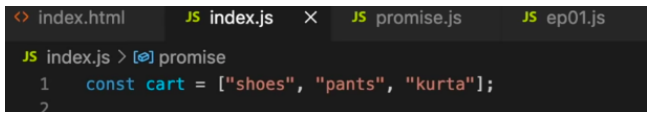
ans. 1) Pending 2) Rejected 3) Fulfilled

Note: Promise objects are immutable. Thus, when we get data in our promise object we can't mutate/change the data.

Q: So what is a Promise?

ans. A promise is an object representing the eventual completion or failure of an asynchronous operation.

→ Solving problem of callback hell with promises:



→ That is a callback hell

- This is a callback hell.

```
createOrder(cart, function (orderId) {
  proceedToPayment(orderId, function (paymentInfp) {
    showOrderSummary(paymentInfp, function () {
      updateWalletBalance();
    });
  });
});
```

const promise: any

→ In order to solve this, we can use promise chaining.

```
createOrder(cart)
  .then(function (orderId) {
    proceedToPayment(orderId);
  })
  .then(function (paymentInfo) {
    showOrderSummary(paymentInfo);
  })
  .then(function (paymentInfo) {
    updateWalletBalance(paymentInfo);
  });
```

Note: We need to make sure we're returning the promise during chaining. That's how we'll get data properly into the chain. So our code will not grow horizontally now.

```
createOrder(cart)
  .then(function (orderId) {
    return proceedToPayment(orderId);
  })
  .then(function (paymentInfo) {
    return showOrderSummary(paymentInfo);
  })
  .then(function (paymentInfo) {
    return updateWalletBalance(paymentInfo);
  });
```

```
createOrder(cart)
  .then((orderId) => proceedToPayment(orderId))
  .then((paymentInfo) => showOrderSummary(paymentInfo))
  .then((paymentInfo) => updateWalletBalance(paymentInfo));
```

→ We can also use arrow f^n's to make code leaner.