Hello, in this video I'll be introducing the topic of promises

and will be giving an overview on what will be covered in this module.

So, what is a promise?

A promise is a container that holds the eventual result of

an asynchronous operation.

Promises are simple.

You start off pending and eventually become either fulfilled or rejected.

Once that occurs a value can be extracted out of the promise.

Promises are becoming the standard way of handling

asynchronous results.

Before promises,

callbacks were commonly used to handle asynchronous results.

The problem with callbacks is that if there are too many of them

chained together, the code becomes a pyramid of callbacks that is hard to

read and hard to manage, especially when handling errors.

On the other hand, promises are much easier to chain together and

have a great way of handling errors.

In this module, we will cover how to create a promise,

how to use a promise, how to chain promises together, and

how to handle multiple promises at the same time.

**Intro to Promises**

**What are Promises?**

Promises are containers for values that are not yet available yet but may eventually become available.

**Why are Promises important?**

Promises are becoming the standard way to handle asynchronous functions in JavaScript.

**What are we going to learn?**

* How to create a Promise
* How to get the result out of a Promise
* How to chain Promises together
* How to handle multiple Promises at the same time

### **Creating and Using Promises**

### **Creating a new Promise**

Sample code to create a promise:

var promise = new Promise(function(resolve, reject) {

//do stuff

var isSuccessful = true;

if (isSuccessful) { /\*if everything is successful\*/

resolve("Success!");

}

else { /\*if something went wrong\*/

reject(Error("Failure."));

}

});

#### ****new Promise()****

The **new Promise()** constructor is called to create a new promise. The constructor takes in a callback function with the arguments **resolve**and **reject**.

var promise = new Promise(function(resolve, reject) {

});

#### ****Resolve()****

The **resolve()** function is used to change the status of the promise from pending to fulfilled. The value that is passed inside the **resolve()** function becomes the fulfillment value of the promise.

Once the **resolve()**function is called, future **resolve()**and **reject()**calls no longer have any effect.

Notice how the **resolve()** method is used to set the fulfillment value of the promise:

resolve("Success!"); //"Success" is set as the fulfillment value of the promise

#### ****Reject()****

The **reject()** function is used to change the status of the promise from pending to rejected. The value that is passed inside the **reject()** function becomes the rejection value of the promise.

Once the **reject()**function is called, future **resolve()**and **reject()**calls no longer have any effect.

The resolve function can take in any object as an argument, but one common practice is to pass in a **Error**object because it provides a more detailed error report.

Notice how a **reject()** is used to send an **Error** object as its reject value:

reject(Error("failure")); //rejection value becomes an Error object

#### ****Promise.resolve() and Promise.reject()****

**Promise.resolve()**is used to return a promise that is already fulfilled. Likewise, the **Promise.reject()** method may be used to return an already rejected promise. Both of these methods can be called outside of the **new Promise()**constructor.

Notice how the **Promise.resolve()** method is used to create an already fulfilled promise:

//A resolved promise with fulfillment value "already resolved"

var resolvedPromise = Promise.resolve("already resolved");

Notice how the **Promise.reject()**method is used to create an already rejected promise:

//A rejected promise with rejected value "already rejected"

var rejectedPromise = Promise.reject("already rejected");

#### ****Resolving another Promise****

If another promise is passed in as an argument to **resolve()** then the new promise takes the fulfillment value of the passed in promise.

Notice how **resolve()** handles another Promise as an argument:

var firstPromise = Promise.resolve("already resolved");

//fullfillment value of secondPromise is "already resolved"

var secondPromise = Promise.resolve(firstPromise);

#### ****Using Promises with Then() and Catch()****

The **then()** and **catch()** methods are used to handle the results of Promises once they have finished pending. The **then()** method is used to handle resolved Promises while the **catch()** method is used to handle rejected Promises. Both of the methods use callback functions. The callback functions should each have one argument representing the Promise result.

Notice how the **then()** and **catch()** methods use callbacks to handle Promise results:

var promise = new Promise(function(resolve, reject) {

//do stuff

var isSuccessful = true;

setTimeout(function(){ //asynchronously process after 5 seconds

if (isSuccessful) { //if everything is successful

resolve('success!');

}

else{ //if something went wrong

reject(Error("failure."))

}

},5000);

});

//promise status changes from pending to resolved after 5 seconds

promise.then(function(val){//val represents the fulfillment value

console.log(val);//logs "success!" since promise resolved

}).catch(function(val){//val represents the rejection value

console.log(val); //doesn't occur since promise never rejected

});

#### Using Promises with Then(onSuccess,onFailure)

The **then()**method can be called with a success callback and a failure callback as an alternative to using the **then()** and **catch()**methods.

Notice how the **then()** method is used with a success and failure callback to handle promise results:

promise.then(function(val){//success callback

console.log(val);

},function(val){//rejection callback

console.log(val);

})

**Chaining Promises**

### **Transforming Values**

#### Calling return within ****then()****

Promise results can be transformed by calling the **return**statement within the **then()** callback. This will cause the **then()** method to return a new Promise with the transformed result.

Notice how a new Promise is created with a transformed result using the **return**statement within the **then()**callback:

var promise = Promise.resolve("hello");

var promise2 = promise.then(function(result) {

console.log(result) //logs "hello"

return result + " world" //adds " world" to the result and sets this as the new fulfillment value of promise2

});

promise2.then(function(result){

console.log(result); //logs "hello world"

});

#### Chaining Transforms

Several transforms can be chained together using multiple **then()** method calls.

Notice how promise results are transformed using multiple **then()** methods calls:

var promise = Promise.resolve([1,2,3,4]);

promise.then(function(result) {

console.log(result) //logs [1,2,3,4]

return result.map(x => x \* x); //squares each value in the array

}).then(function(result2){

console.log(result2) //logs [1,4,9,16]

return result2.filter( x => x > 10); //filters out elements that are not larger than 10

}).then(function(result3){

console.log(result3) //logs [16]

return result3.toString() + "!!"; //converts result3 to a string and adds "!!"

}).then(function(result4){

console.log(result4) //logs "16!!"

return result4; //returns a promise with "16!!" as the fulfillment value

}).catch(function(error){

console.log(error)

});

### **Sequencing Asynchronous Operations**

#### Returning a Promise within ****then()****

Returning another Promise within a **then()** callback will cause the **then()**method to return the returned Promise.

Notice how returning a Promise within a **then()** callback creates a new promise with the returned promise's result:

var promise = Promise.resolve("hello");

var promise2 = promise.then(function(result) {

console.log(result) //logs "hello"

return Promise.resolve("12345") //causes then() to return a promise with a fulfillment value of "12345"

});

promise2.then(function(result){

console.log(result); //logs "12345"

});

#### Sequencing Asynchronous Operations

Asynchronous Operations can be sequenced by returning Promises within **then()** callbacks.

Imagine that there are three asynchronous functions called getRandomNumber(), getNameFromNumber() and getAgeFromName() that return Promises.

The functions do the following:

* getRandomNumber() - asynchronously returns a random number
* getNameFromNumber - takes in a number and asynchronously returns a name
* getAgeFromName - takes in a name and asynchronously returns an age

If we wanted to first call getRandomNumber() to get an number, then call getNameFromNumber() to get a name from that number, and then lastly call getAgeFromName() on the returned name to get an age then we would have to sequence them correctly.

If they were normal synchronous functions then it would be simple and would look like this:

var number = getRandomNumber();

var name = getNameFromNumber(number);

var age = getAgeFromName(name);

However, since the functions are asynchronous, the number variable may be undefined by the time getNameFromNumber() is called and the name variable may be undefined by the time getAgeFromName() is called.

Thus, we need to do the following to sequence them correctly:

//getRandomNumber() returns a promise containing a random number

getRandomNumber().then(function(result) {

console.log(result) // 42

return getNameFromNumber(result); //returns a promise containing a string representing a name

}).then(function(result2){

console.log(result2) //"Bob"

return getAgeFromName(result2); //returns a promise containing a number representing an age

}).then(function(result3){

console.log(result3) //21

}.catch(function(error){

console.log(error)

});

If any of the **then()** functions returns a rejected Promise then the **catch()** method will handle the rejected result.

#### Chaining Promises vs Continuation Passing Style(CPS)

Lets see how Promises compare with CPS when trying to chain asynchronous operations.

Chaining Asynchronous Operations using the Continuation Passing Style:

getRandomNumber(function(number)){

console.log(number); //logs 42

getNameFromNumber(number, function(name)){

console.log(name) //logs 'Bob'

getAgefromNumber(age, function(age)){

console.log(age) //logs 21

//do something with age

},

function(error){

console.log(error);

}

},

function(error){

console.log(error);

}

},function(error){

console.log(error);

});

Chaining Asynchronous Operations using Promises:

//getRandomNumber() returns a promise containing a random number

getRandomNumber().then(function(result) {

console.log(result) // 42

return getNameFromNumber(result); //returns a promise containing a string representing a name

}).then(function(result2){

console.log(result2) //"Bob"

return getAgeFromName(result2); //returns a promise containing a number representing an age

}).then(function(result3){

console.log(result3) //21

}.catch(function(error){

console.log(error)

});

As you can see, it is difficult to make changes to a chain of asynchronous operations using CPS, especially since there has to be a callback for both the success and failure cases for each asynchronous call.

Promises allow asynchronous operations to be chained in a much more maintainable way.

**Handling Multiple Promises**

### **Promise.all()**

The **Promise.all()** method is used to process multiple Promises at the same time. The method takes in an array of Promises and then waits for them to all to resolve. Once they have all finished resolving, an array of results can be obtained by using the **then()** method. If any of the Promises reject, then the **Promise.all()** method will return the first rejected Promise.

Notice how the **Promise.all()**method is used to handle multiple Promises at the same time:

var promise1 = Promise.resolve('hello');

var promise2 = Promise.resolve({age:2,height:188})

var promise3 = 42; //normal values work with Promise.all() too

Promise.all([promise1,promise2,promise3]).then(function(result) {

console.log(result) //logs the array ["hello",{age:2,height:188},42]

}).catch(function(error){

console.log(error)

});

Notice how **Promise.all()** method call rejects when one of the Promises that it is processing rejects:

var promise1 = Promise.resolve('hello');

var promise2 = Promise.resolve({age:2,height:188})

var promise3 = Promise.reject('failure.'); //rejected promise

Promise.all([promise1,promise2,promise3]).then(function(result) {

console.log(result) //doesn't occur since promise3 rejected

}).catch(function(error){

console.log(error) //logs 'failure.'

});

### **Promise.Race()**

The **Promise.race()** method takes in an array of promises and takes the result of the promise that rejects or resolves the fastest. Like normal promises, the **then()** and **catch()** methods are used to retrieve the results of the fastest promise.

The **Promise.race()**method can be used to choose the quickest source when there are two similar sources of the same data.

Notice how the **Promise.race()** method is used to take the result of the faster promise:

var promise1 = new Promise(function(resolve,reject){

setTimeout(function(){

resolve("finished in two seconds");

},2000) //returns a resolved promise after 2 seconds

});

var promise2 = new Promise(function(resolve,reject){

setTimeout(function(){

resolve("finished in five seconds");

},5000) //returns a resolved promise after 5 seconds

});

Promise.race([promise1,promise2]).then(function(result) {

console.log(result) // logs "finished in two seconds" because promise1 resolved first

}).catch(function(error){

console.log(error)

});

The **Promise.race()**method can also be used to limit the amount of time promises have to resolve by including a promise that is forced to reject after a given amount of time.

Notice how the **Promise.race()** method is used to limit the amount of time a Promise has to resolve:

var promiseResolveTenSeconds = new Promise(function(resolve,reject){

setTimeout(function(){

resolve("finished in ten seconds");

},10000) //returns a resolved promise after 10 seconds

});

var promiseRejectFiveSeconds = new Promise(function(resolve,reject){

setTimeout(function(){

reject("error: promise took longer than 5 seconds to resolve");

},5000) //returns a rejected promise after 5 seconds

});

Promise.race([promiseResolveTenSeconds,promiseRejectFiveSeconds]).then(function(result) {

console.log(result) // never occurs because promiseRejectFiveSeconds rejected

}).catch(function(error){

console.log(error) // logs "error: promise took longer than 5 seconds to resolve"

});