**Promises**

An *asynchronous operation* is one that allows the computer to “move on” to other tasks while waiting for the asynchronous operation to complete.

1. **Introduction**

Promises are objects that represent the eventual outcome of an asynchronous operation. A Promise object can be in one of three states:

A picture containing graphical user interface

Description automatically generated

* **Pending**: The initial state— the operation has not completed yet.
* **Fulfilled**: The operation has completed successfully and the promise now has a *resolved value*. For example, a request’s promise might resolve with a JSON object as its value.
* **Rejected**: The operation has failed and the promise has a reason for the failure. This reason is usually an Error of some kind.

🡪 All promises eventually settle, enabling us to write logic for what to do if the promise fulfills or if it rejects.

1. **Constructing a Promise Object**

To create a new Promise object, we use the new keyword and the Promise constructor method:

const executorFunction = (resolve, reject) => { };

const myFirstPromise = new Promise(executorFunction);

The Promise constructor method takes a function parameter called the executor function which runs automatically when the constructor is called.

The executor function generally starts an asynchronous operation and dictates how the promise should be settled.

The executor function has two function parameters, usually referred to as the resolve() and reject() functions. The resolve() and reject() functions aren’t defined by the programmer. When the Promise constructor runs, JavaScript will pass its own resolve() and reject() functions into the executor function.

* resolve is a function with one argument. Under the hood, if invoked, resolve() will change the promise’s status from pending to fulfilled, and the promise’s resolved value will be set to the argument passed into resolve().
* reject is a function that takes a reason or error as an argument. Under the hood, if invoked, reject() will change the promise’s status from pending to rejected, and the promise’s rejection reason will be set to the argument passed into reject().

For example:

Text

Description automatically generated

* We declare a variable myFirstPromise
* myFirstPromise is constructed using new Promise() which is the Promise constructor method.
* executorFunction() is passed to the constructor and has two functions as parameters: resolve and reject.
* If someCondition evaluates to true, we invoke resolve() with the string 'I resolved!'
* If not, we invoke reject() with the string 'I rejected!'

1. **The Node setTimeOut() Function**

Knowing how to construct a promise is useful, but most of the time, knowing how to *consume*, or use, promises will be key. Rather than constructing promises, you’ll be handling Promise objects returned to you as the result of an asynchronous operation. These promises will start off pending but settle eventually.

Moving forward, we’ll be simulating this by providing you with functions that return promises which settle after some time. To accomplish this, we’ll be using setTimeout(). setTimeout() is a Node API (a comparable API is provided by web browsers) that uses callback functions to schedule tasks to be performed after a delay. setTimeout() has two parameters: a callback function and a delay in milliseconds.

Text

Description automatically generated

🡪 The delayedHello will be run **in at least 2 seconds.**

The reason is that this delay is performed asynchronously (the rest of the program won’t stop executing during the delay).

Asynchronous JavaScript uses *the event-loop*. After two seconds, delayedHello() is added to a line of code waiting to be run. Before it can run, any synchronous code from the program will run. Next, any code in front of it in the line will run. This means it might be more than two seconds before delayedHello() is actually executed.

After waittime 🡪 Call stack 🡪 Codes in front of setTimeOut() in task queue 🡪 Code in setTimeOut()

Let’s see how we can use setTimeOut() to construct asynchronous promises:

Text

Description automatically generated

In the example code, we invoked returnPromiseFunction() which returned a promise. We assigned that promise to the variable prom. Similar to the asynchronous promises you may encounter in production, prom will initially have a status of pending.

1. **Consuming Promises using .then()**

After a promise settles, we want to tell the computer what to do afterwards. This is where we use the .then() method of the Promise object. This is like saying: *“I have a promise, when it settles,****then****here’s what I want to happen…”*.

A screenshot of a cell phone

Description automatically generated with medium confidence

.then() is a higher-order function— it takes two callback functions as arguments. We refer to these callbacks as ***handlers***. When the promise settles, the appropriate handler will be invoked with that settled value.

* The first handler, sometimes called onFulfilled, is a ***success*** *handler*, and it should contain the logic for the promise resolving.
* The second handler, sometimes called onRejected, is a ***failure*** *handler*, and it should contain the logic for the promise rejecting.

We can invoke .then() with one, both, or neither handler.

One important feature of .then() is that it always returns a promise.

“.*then() itself returns a promise, which will be completed with the result of the function passed to it*.”

1. **Success and Failure Callback Functions**

Text

Description automatically generated

* prom is a promise which will randomly either resolve with 'Yay!' or reject with 'Ohhh noooo!'.
* When prom resolves, handleSuccess/handleFailure is invoked with prom’s resolved/rejected values 🡪 These resolved/rejected values are passed to the handler functions.
* We pass two handler functions to .then(). The first will be invoked with 'Yay!' if the promise resolves, and the second will be invoked with 'Ohhh noooo!' if the promise rejects.

We can have another example with the use of setTimeout():

Text

Description automatically generatedText

Description automatically generated

🡪 We used setTimeout() to ensure that the checkInventory() promise settles asynchronously.

1. **Using catch() with Promises**

Remember that .then() will **return a promise** with the same settled value as the promise it was called on, if no appropriate handler was provided

🡪 This implementation allows us to separate our resolved logic from our rejected logic. Instead of passing both handlers into one .then(), we can chain a second .then() with a failure handler to a first .then() with a success handler and both cases will be handled.

Text

Description automatically generated

*If the promise from prom rejects, the first .then() will return a promise with the same rejection reason as the original promise, which is passed into the second .then()*

To create a more readable code, we can use a different promise function .catch(). This function takes only ONE parameter, the onRejected funtion 🡪 accomplishes the same thing as using a .then() with **only failure handler.**

Text

Description automatically generated

1. **Chaining multiple Promises \*\***

One common pattern we’ll see with asynchronous programming is multiple operations which depend on each other to execute or that must be executed in a certain order. We might make request to a database and use the data returned to make another request and so on 🡪 needs chaning promises.

Text

Description automatically generatedThis process of chaining promises together is called *composition*. For example:

* firstPromiseFunction() 🡪 a promise
* .then() as the success handler, returning a new promise - the result of invoking secondPromiseFunction() with the first promise’s resolved value (firstResolveVal).
* .then() to handle logic for the second promise settling (logging to the console)

\*\* We had to return, to ensure that the return value of the first .then() was our second promise (secondResolveVal), not the default return of a new promise with the same settled value as the initial (firstResolveVal)

1. **Avoiding common mistakes**

**Nesting promises instead of chaining them** 🡪 We have to separate them. Else, we are nesting the logic of one promise inside the other

**Forgetting to return** 🡪 the second .then() should take the value returned by the secondPromiseFunction, or the secondResolveVal, not the firstResolveVal that is the default value of return of the first .then()

1. **Promise.all()**

Promise composition is a great way to deal with multiple promises that depends on each other. However, sometimes we just **don’t care about the order,** and want to maximize efficiency by allowing *concurrency*, or multiple asynchronous operations happening together. With promises, we can do this with the function Promise.all().

Promise.all() accepts an array of promises as its argument and returns a single promise. That single promise will settle in one of two ways:

* If every promise in the argument array resolves, the single promise returned from Promise.all() will resolve with an array containing the resolve value from each promise in the argument array.
* If any promise from the argument array rejects, the single promise returned from Promise.all() will immediately reject with the reason that promise rejected. This behavior is sometimes referred to as *failing fast*.

🡪 As soon as the first promise in the array rejects, the promise returned from Promise.all() will reject with that reason.

Text

Description automatically generated