# JavaScript Promises with Node.js

In this article, I explain how to use JavaScript promises by the means of short Node.js example programs.

I will also explain what asynchronous tasks are and how they are implemented in JavaScript with promises and callbacks.

Since, I’m using Node.js programs as examples, I will also briefly explain what Node.js is and how it works.

Finally, throughout the article, I compare the use of promises to the more traditional use of callbacks for implementing asynchronous tasks in JavaScript.

## Contents

* [**What are Promises? What are Asynchronous Tasks?**](https://itnext.io/javascript-promises-with-node-js-e8ca827e0ea3#4029)
* [**What is Node.js?**](https://itnext.io/javascript-promises-with-node-js-e8ca827e0ea3#199b)
* [**Simple Asynchronous Tasks with Promises**](https://itnext.io/javascript-promises-with-node-js-e8ca827e0ea3#b7e6)
* [**Chained Asynchronous Tasks with Promises**](https://itnext.io/javascript-promises-with-node-js-e8ca827e0ea3#9482)
* [**Creating Promises**](https://itnext.io/javascript-promises-with-node-js-e8ca827e0ea3#cc7d)

## What are Promises? What are Asynchronous Tasks?

Promises and callbacks are two mechanisms to implement **asynchronous tasks**in JavaScript. A task is asynchronous if the caller just “kicks off” the task and then goes on doing other work while the task runs. When the task completes, the caller gets notified, so that it can interrupt its current execution and do whatever there is to do with the result of the task.

This stands in contrast to **synchronous tasks**, in which case the caller just waits until the task is finished, and only then resumes execution.

Typically, programs contain both, synchronous and asynchronous, tasks. Naturally, long-lasting “slow” tasks are good candidates for asynchronous tasks (very quick tasks can be implemented synchronously). The slowest tasks are generally input/output (I/O) tasks. **I/O tasks** are tasks that access external devices, such as a hard disk or the network. In JavaScript, I/O tasks are thus typically implemented asynchronously.

When you start an asynchronous task in a program, you must somehow define what to do when this task finishes. This is done traditionally by the means of a **callback function** that you pass as an argument to the function that starts the asynchronous task. This function will then be executed when the task completes.

More recently, there is another way to define the actions to take after an asynchronous task completes: **promises**. Promises have been introduced in the [ECMA-262 Standard, 6th Edition](http://www.ecma-international.org/ecma-262/6.0/) (ES6) that has been published in June 2015.

The goal of this article is to explain the use of promises by examples. Note that promises can be used as an alternative to callback functions, and everything that can be done with promises can also be done with callback functions. But promises are more specialised and allow for cleaner code. This will also become apparent in this article when we compare the use of promises to the use of callbacks.

Asynchronicity lies at the heart of JavaScript. JavaScript **runtime environments** are specialised for the execution of asynchronous tasks. They have special facilities for the handling of asynchronous tasks. This also applies to the particular JavaScript runtime environment of **Node.js**. But what is Node.js exactly?

## What is Node.js?

Simply said, Node.js is a *standalone* **runtime environment** for JavaScript. That means, Node.js can execute JavaScript code as an independent process on your computer. We can say that Node.js is to JavaScript what the JVM is to Java bytecode, or what the Python interpreter is to Python.

What’s special about this is the *standalone* part. Because, traditionally JavaScript engines where exclusively embedded into web browsers. That means, you could execute JavaScript code only in the context of a website that is being displayed in a web browser. Node.js was one of the first JavaScript engines that was independent of a web browser and runs in its own process.

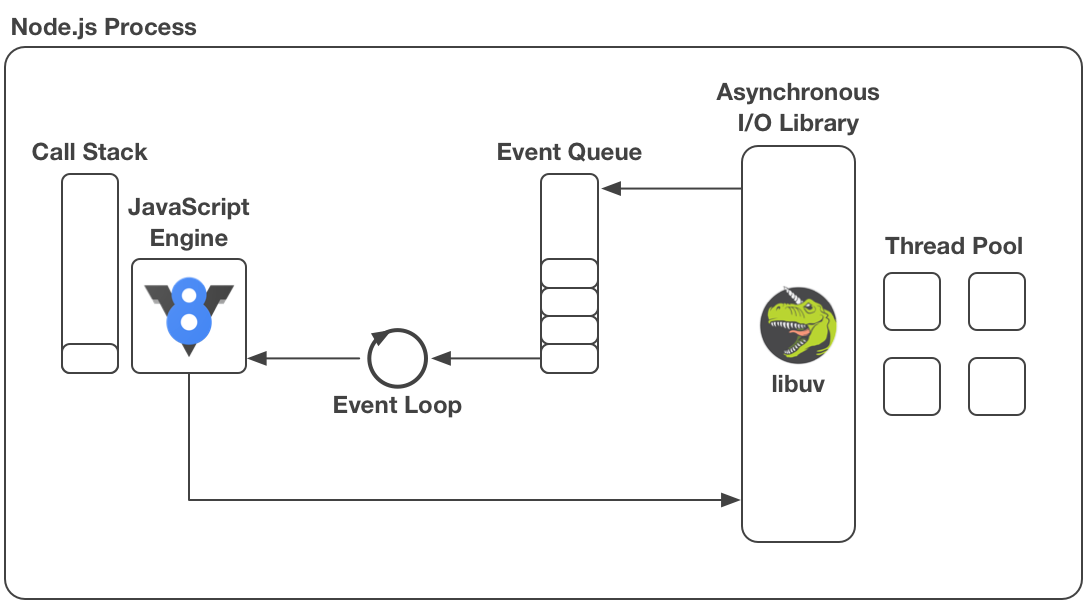
Node.js makes it possible to use JavaScript for **general-purpose applications**. Today the most common type of applications that are built with Node.js are web servers. But in principle Node.js can be used to build *any* type of applications.

## How Does It Work?

Node.js has the following three main ingredients:

* **JavaScript engine**
* **Asynchronous I/O library**
* **Event loop**

In the following, I’m going to explain each of these concepts. Here is already an overview of the internal Node.js architecture:



Internal workings of Node.js.

**JavaScript engine:**

A JavaScript engine is a software that reads JavaScript code and makes it execute on the machine.

Most JavaScript engines today perform **just-in-time (JIT) compilation**, that is, JavaScript statements are compiled to machine code while the program is being executed. That makes them something between classical ahead-of-time (AOT) compilers (such as gcc for C) and classical interpreters (such as the interpreters for Ruby or Python). Here is a [video](https://www.youtube.com/watch?v=p-iiEDtpy6I) that explains how JavaScript engines work.

The important thing to remember is that the job of a JavaScript engine is to take a statement or function of JavaScript and make it execute on the machine.

Web browsers contain a JavaScript engine and so does Node.js. The JavaScript engine used by Node.js is[**Chrome V8**](https://developers.google.com/v8/) (or simply V8) developed by Google.

**Asynchronous I/O library:**

A JavaScript engine is not enough for a complete runtime environment. Slow operations, such as a disk access or network request, that are implemented asynchronously are not executed by the JavaScript engine itself, but they are handed over for execution to a specialised asynchronous I/O library. The library used by Node.js is called [**libuv**](https://libuv.org/).

libuv maintains a **pool of worker threads** on which I/O tasks received from the JavaScript engine are executed. The default size of the thread pool is 4, the maximum size is 128, and it can be changed at startup time of Node.js by setting the UV\_THREADPOOL\_SIZE environment variable (see [docs](http://docs.libuv.org/en/v1.x/threadpool.html)).

Note that the threads in the thread pool *may* block (for example, waiting for a disk access), but this doesn’t block the JavaScript engine which runs in its own thread. If all the threads in the thread pool are busy and new tasks from the JavaScript engine arrive, then these new tasks are simply queued up by libuv and they will be processed as soon as a worker thread becomes free.

When the JavaScript engine passes an I/O task to libuv, it also registers the **callback** that is associated with this task with the runtime. This callback will then be executed when the task completes (we will see in a moment how this happens in detail). That means, each task passed to libuv must have a callback associated with it. This callback is the callback function that you pass as an argument to asynchronous functions.

Now let’s see what’s going to happen with this callback when libuv completes a task.

**Event loop:**

The event loop works in tandem with the **event queue**. The event queue is simply a queue of JavaScript functions. For clarity, let’s call the functions in the event queue *events*. When libuv completes a task, then the runtime puts the associated callback in the event queue.

Note that alternative names that are commonly used for the event queue are *task queue* or *message queue*. But they are all the same thing.

Now to the **event loop**: the event loop sits between the event queue and the JavaScript engine. Simply said, the job of the event loop is to “shovel” events from the event queue into the JavaScript engine for execution.

But the event loop does this not blindly, but according to the following rules:

* If the JavaScript engine is currently already executing an event, then the event loop waits until the JavaScript engine completes the execution of this event.
* As soon as the JavaScript engine is idle, the event loop passes the event at the head of the event queue to the JavaScript engine. If the event queue is currently empty but there are still pending asynchronous tasks, then the event loop just waits until a new event is put into the event queue.

The first of these points is often explained with the **call stack**. Like most runtimes, Node.js also maintains a call stack to facilitate the execution of functions. When a function is being called, a “stack frame”, consisting of the return address and local variables of the function, is pushed onto the stack, and when the function returns, the stack frame is removed from the stack.

Whenever the call stack is not empty, it means that the JavaScript engine is still executing an event. When the call stack is empty, it means that the JavaScript engine completed executing its last event (that was previously passed to it by the event loop), which means that the JavaScript engine is ready to receive the next event from the event loop.

From this fact come the expressions that Node.js always “clears the stack” before processing new events, and that all events [“run to completion”](https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop#Run-to-completion) before anything else is executed.

Here is a good [**video**](https://www.youtube.com/watch?v=8aGhZQkoFbQ) that explains how the event loop works.

Note that the above video is not about Node.js specifically, but about JavaScript runtimes as found in web browsers. However, Node.js is actually very similar to the JavaScript runtime of a web browser. The biggest difference is probably that Node.js uses libuv, whereas web browser runtimes use browser-provided APIs for the asynchronous I/O tasks. The event loop works pretty much the same in Node.js and in web browser JavaScript runtimes.

## How is the Main Program Executed?

So far we have talked about “events” which are callbacks and which are put into the event queue when the corresponding asynchronous tasks complete. But what about the “main code” of a program? Or a program that uses no asynchronous tasks and callbacks at all?

For example, the following program:

var n = 1 + 1;  
console.log("The answer is " + n);

How is it executed? Well, the main code of a program can be seen as the body of a special function main(), and this function is put as the very first event into the event queue when the program is started. Consequently, the event loop will pass it immediately to the JavaScript engine, and the JavaScript engine will execute it as its first event.

This also explains an important observation that is often made with JavaScript. Consider the following program:

myLog("Start of main");

setTimeout(() => myLog("I'm the setTimeout callback"), 0);

for (i = 0; i < 1000000000; i++) {}

myLog("End of main");

And its output:

$ node app.js  
12:26:35.423 Start of main  
12:26:38.662 End of main  
12:26:38.665 I'm the setTimeout callback

You can see that even though we set a delay of 0 milliseconds for the setTimeout callback, this callback is executed only 3 seconds later, after all the other statements in the main program have been executed.

The reason for this is that the setTimeout callback results in a new event that is put into the event queue after the main event has already started to be executed by the JavaScript engine (note that setTimeout creates an asynchronous task for libuv which will complete when the set delay expires, thus creating a new event, consisting of the callback function, that is put into the event queue).

The callback event will be in the event queue almost immediately after setTimeout is called in the main program (the delay is set to 0 milliseconds). It will be sitting there at the head of the event queue, waiting to be executed. But the event loop can’t do anything about it. The JavaScript engine is still executing the main event, and the event loop has no choice but waiting until the JavaScript engine completes execution of the main event (with its lengthy for loop), before it can pass the setTimeout callback event to the JavaScript engine.

This is the reason that the setTimeout is not a *guaranteed delay*, but a *minimum delay*. This is also explained [here](https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop#Adding_messages).

After this lengthy discussion about how Node.js and the event loop works, let’s return to the topic of the article: **promises**.

## Simple Asynchronous Tasks with Promises

Asynchronous functions return promises. You then define what to do when the task completes by supplying a callback function to this promise. This contrasts with the traditional use of callbacks where you pass the callback function directly to the asynchronous function.

We will use an example the asynchronous API functions of the amqplibmodule. amqplib is a client library for AMQP message queuing servers, such as [RabbitMQ](http://www.rabbitmq.com/).

This library has the nice property that all the API functions are available as a promise-based version as well as a callback-based version. So, we can conveniently compare the use of promises to the use of callbacks.

Our first program listed below simply tries to establish a connection with an AMQP server on the local machine:

var amqp = require('amqplib');

amqp.connect("amqp://localhost").then(onSuccess).catch(onFailure);

function onSuccess(result) {  
 myLog(result.toString());  
}

function onFailure(err) {  
 myLog(err.toString());  
}

myLog("I'm here");

Note that the definition of the myLog function is omitted in the above listing, but it can be found at the end of the article in the [appendix](https://itnext.io/javascript-promises-with-node-js-e8ca827e0ea3#df35).

The amqp.connect function returns a promise. We then call then on this promise, passing it the callback function for the case the connection establishment succeeds, and then we call the catch function, passing it another callback function for the case the connection establishment fails. I will explain the use of then and catch in more detail later.

By the way, a documentation of promises can be found [here](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise).

Now, without having an AMQP sever running, let’s execute this program:

$ node app.js  
12:11:13.580 I'm here  
12:11:13.595 Error: connect ECONNREFUSED 127.0.0.1:5672

We can see two things in the output. First, the connection couldn’t be established (the onFailure function was called, and not the onSuccessfunction). This is simply because we have no AMQP server running on our local machine. Later, we will start a RabbitMQ server and then run the program again.

Second, we can see that the myLog("I'm here") statement is executed 15 milliseconds before the myLog(err.toString()) statement that makes up the body of the onFailure function. Let’s see why this sequence of events is how it is.

## What’s Going On?

Node.js uses a single main thread that runs the JavaScript engine. As mentioned above, synchronous tasks are directly executed by the main thread. Asynchronous tasks, on the other hand, are passed by the main thread to libuv, the asynchronous I/O library. After that, the main thread goes on executing the next statement in the program.

At the same time, worker threads execute the tasks that were posted to libuv. As soon as a task completes, the associated callback is put as a new event into the event queue, from where it will be eventually passed to the JavaScript engine in the main thread by the event loop.

Creating a promise basically posts an asynchronous task to libuv, and the then function of a promise registers a callback with the asynchronous task associated with this promise.

So, when we execute the above program, the following sequence of steps occurs:

1. The main thread executes amqp.connect. This causes an appropriate *connect* task to be passed to libuv for asynchronous execution. Thereafter, amqp.connect completes and returns a promise.
2. ***A worker thread from the libuv thread pool may now start executing the connect task.***
3. The main thread executes the then function of the Promise returned by amqp.connect, and registers the supplied onSuccess function as a callback for the *connect* task (for the case that the task *succeeds*). The then function returns another Promise.
4. The main thread executes the catch function of the Promise returned by then, and registers the supplied onFailure function as another callback for the *connect* task (for the case that the task *fails*). The catch function returns another promise (but this one is not further used in our example).
5. The main thread executes myLog("I'm here") (printing I'm here to the console).
6. ***For around 15 milliseconds, the main thread is idle and the event queue is empty, so nothing happens.***
7. A worker thread completes the *connect* task (it failed), which causes the runtime to put the onFailure function into the event queue.
8. The main thread executes the onFailure function (printing Error connect ECONNREFUSED ... to the console).
9. The Node.js process terminates.

## A Note About “catch" and "then"

The catch(onFailure) function is just syntactic sugar for then(undefined, onFailure). That is, catch is just another then function with the first parameter set to undefined (see [docs](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/catch)).

Having catch out of the way, we can talk about then and the ways it can be called.

The then function takes two arguments, whereof the second argument is optional. The first argument is a callback function to be called when the promise, upon which then is called, is **“fulfilled”**, and the second (optional) argument is callback function to be called when the promise is **“rejected”**(see [docs](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then)).

Note that both of these function arguments are functions with a single parameter like the ones we defined above as onSuccess and onFailure.

So, a typical invocation of then looks like this:

then(onSuccess, onFailure)

But it can also be called like this:

then(onSuccess)

Or like this:

then(undefined, onFailure)

As we have seen, this last invocation is equivalent to this:

catch(onFailure)

The first case, then(onSuccess, onFailure), is pretty clear: if the task in the promise *succeeds*(i.e. the promise is “fulfilled”), thenonSuccess is called, and if the task *fails*(i.e. the promise is “rejected”), then onFailure is called.

Similarly, in the second case, then(onSuccess), if the promise is fulfilled, then onSuccess is called. But what if the promise is rejected? Then actually no callback code is executed (the promise rejection is not handled). As we will see later, this can actually be a desired behaviour if then functions are *chained*.

In the third case, then(undefined, onFailure), which is equivalent to catch(onFailure), if the promise is rejected, then onFailure is called. But what if the promise is fulfilled? Again, then no callback code is executed (the Promise fulfilment is not handled). Again, this can be useful if then functions are chained.

In the above listing, we had the following line:

amqp.connect("amqp://localhost").then(onSuccess).catch(onFailure);

With this new knowledge, we see that the following line is equivalent:

amqp.connect("amqp://localhost").then(onSuccess).then(undefined, onFailure);

And we could also achieve the same behaviour with a single invocation of then like this:

amqp.connect("amqp://localhost").then(onSuccess, onFailure);

The [docs](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then#Chaining) recommend to use then(onSuccess) and catch(onFailure) rather than then(onSuccess, onFailure).

**Allow Promise Fulfillment**

Start a RabbitMQ server on the local machine with rabbitmq-server and then run the program again:

$ node app.js  
12:14:59.448 I'm here  
12:14:59.505 [object Object]

Now the connection succeeded. In contrast to before, now the onSuccess, rather than onFailure, function was called, at least 57 milliseconds after the connection establishment was initiated with the call to amqp.connect.

If you navigate now to the RabbitMQ Management UI on [http://localhost:15672/#/connections](http://localhost:15672/#/connections,), you should see the open connection there.

The [object Object] that is printed in the log output is actually an amqplibobject that represents an open connection to an AMQP server, and it can be used for further API functions of amqplib, for example, for creating a [channel](http://www.squaremobius.net/amqp.node/channel_api.html#model_createChannel).

## With Anonymous Functions

In the above examples, we use the explicitly named functions onSuccess and onFailure as the arguments for the then and catch functions.

It is more common (even though it’s often less readable) to use anonymous functions in these situations. In this case, the program would look like this:

var amqp = require('amqplib');

amqp.connect("amqp://localhost")  
 .then(result => myLog(result.toString()))  
 .catch(err => myLog(err.toString()));

myLog("I'm here");

This is exactly equivalent to the above version with named callback functions.

## Promises vs. Callbacks

The amqplib module has a promise-based and a callback-based version of every function in the API. To import the callback-based function, we just have to require amqplib/callback\_api instead of amqplib.

In the callback-based version, the connect function takes two arguments: an AMQP URI and an [error-first callback](http://fredkschott.com/post/2014/03/understanding-error-first-callbacks-in-node-js/) function. Let’s rewrite the above program with the callback-based API function:

var amqp = require('amqplib/callback\_api');

amqp.connect("amqp://localhost", callback);

function callback(err, result) {  
 if (err)  
 myLog(err.toString());  
 else  
 myLog(result.toString());  
}

myLog("I'm here");

You can run this program with and without a running RabbitMQ server: the behaviour is exactly the same as in the initial Promise-based version.

Now, for completeness, let’s also rewrite the callback-based version with anonymous, rather than named, functions:

var amqp = require('amqplib/callback\_api');

amqp.connect("amqp://localhost", (err, result) => {  
 if (err)  
 myLog(err.toString());  
 else  
 myLog(result.toString());  
});

myLog("I'm here");

We can already see here that the callback-based version is slightly less terse than the promise-based version. But this difference will be accentuated for more complex task, as we will see in the next section.

**Chained Asynchronous Tasks with Promises**

Promises, and especially promise chains, become really useful if we have to execute a chain of asynchronous tasks with the restriction that a task can only start as soon as the previous task completed. This results in nested callbacks, because we have definitions of callbacks within other callbacks.

For example, for completely setting up a RabbitMQ server, we might have execute the following tasks.

1. Establish a *connection* to the RabbitMQ server
2. When (1) is done, create a *channel*on this connection
3. When (2) is done, create a *queue*on this channel

Here is how we could implement this with chained promises:

var amqp = require('amqplib');

amqp.connect("amqp://localhost")  
 .then(onConnSuccess)  
 .then(onChannelSuccess)  
 .then(onQueueSuccess)  
 .catch(onFailure);

function onConnSuccess(connection) {  
 myLog("Connection: " + connection.toString());  
 return connection.createChannel()  
}

function onChannelSuccess(channel) {  
 myLog("Channel: " + channel.toString());  
 return channel.assertQueue("my\_queue");  
}

function onQueueSuccess(queue) {  
 myLog("Queue: " + queue.toString());  
}

function onFailure(err) {  
 myLog(err.toString());  
}

myLog("I'm here");

As you can see, there is a success callback function for each task, and the first two of these functions initiate a new asynchronous task and return the resulting Promise into the Promise chain.

The then are internally set up in a way that allows the callback functions to return a promise, which is applied to the subsequent then in the chain, and so on. If the callback function returns a non-promise value, then the current promise is “resolved”, that is, fulfilled, with this value. Details about the return value of then can be found in the [**docs**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then#Return_value). All in all, this sophisticated design allows for promise chains as shown above.

At the end of the chain there is a single catch(onFailure) function (which you know is the same as then(undefined, onFailure)). This function is invoked as soon as any of the previous Promises in the chain is rejected. This means, if a promise early in the chain is rejected, then the subsequent thenfunctions are not executed, but execution jumps directly to the catch. This is why, as mentioned before, for chaining it is useful if some then handle only fulfillments, but not rejections, and others (the catch) handle only rejections but not fulfillments.

Let’s see how this works in practice. If we have no RabbitMQ server running and run the above program, we get the following:

$ node app.js  
18:26:28.509 I'm here  
18:26:28.548 Error: connect ECONNREFUSED 127.0.0.1:5672

The only callback that was executed here is onFailure with the error object for the connection Promise. As expected, none of the then callbacks is executed.

Now, let’s start a RabbitMQ server with rabbitmq-server and run the program again:

$ node app.js  
18:35:35.337 I'm here  
18:35:35.362 Connection: [object Object]  
18:35:35.368 Channel: [object Object]  
18:35:35.371 Queue: [object Object]

As we can see, now the callback functions of all the then are executed in order, creating a connection, then a channel, and then a queue. The catch, on the other hand, is not executed, because none of the tasks failed.

Go to check your RabbitMQ server on [http://localhost:1567](http://localhost:15672/) and you should see that a connection, a channel, as well as a queue, have been created.

## Promises vs. Callbacks

Everything that can be done with promises can also be done with callbacks. So let’s see how we could write the above program with callbacks (using the “callback version” of the amqplib API):

var amqp = require('amqplib/callback\_api');

amqp.connect("amqp://localhost", connCallback);

function connCallback(err, connection) {  
 if (err)  
 myLog(err.toString());  
 else {  
 myLog("Connection: " + connection.toString());  
 connection.createChannel(channelCallback);  
 }  
}

function channelCallback(err, channel) {  
 if (err)  
 myLog(err.toString());  
 else {  
 myLog("Channel: " + channel.toString());  
 channel.assertQueue("my\_queue", null, queueCallback);  
 }  
}

function queueCallback(err, queue) {  
 if (err)  
 myLog(err.toString());  
 else {  
 myLog("Queue: " + queue.toString());  
 }  
}

myLog("I'm here");

The promise-based version is more terse than the callback-based version, and I think it can be argued that it is more readable too.

## With Anonymous Functions

Let’s see how both, the promise-based and the callback-based versions, look like when we use anonymous functions.

**Promise version:**

var amqp = require('amqplib');

amqp.connect("amqp://localhost")  
 .then(connection => {  
 myLog("Connection: " + connection.toString());  
 return connection.createChannel()  
 })  
 .then(channel => {  
 myLog("Channel: " + channel.toString());  
 return channel.assertQueue("my\_queue");  
 })  
 .then(queue => {  
 myLog("Queue: " + queue.toString());  
 })  
 .catch(err => {  
 myLog(err.toString());  
 });

myLog("I'm here");

## Callback version:

var amqp = require('amqplib/callback\_api');

amqp.connect("amqp://localhost", (err, connection) => {  
 if (err)  
 myLog(err.toString());  
 else {  
 myLog("Connection: " + connection.toString());  
 connection.createChannel((err, channel) => {  
 if (err)  
 myLog(err.toString());  
 else {  
 myLog("Channel: " + channel.toString());  
 channel.assertQueue("my\_queue", null, (err, queue) => {  
 if (err)  
 myLog(err.toString());  
 else {  
 myLog("Queue: " + queue.toString());  
 }  
 });  
 }  
 });  
 }  
});

myLog("I'm here");

Here the advantage of promises over callback functions becomes pretty apparent.

With callbacks we have to define nested anonymous callback definitions and we end up in “callback hell”. Furthermore, we have to check for errors with if/else statements what convolutes the readability further.

With promises, on the other hand, we have a constant nesting depth for all anonymous functions and errors are handled as elements of the chain (by catch functions).

This increased readability and reduced error-proneness are the main advantage of promises over callbacks.

## Creating Promises

All the above discussions were about *using* promises. In each case, the promise was created by an API function, and we just use it.

However, if you have to create yourself an asynchronous function with promise-capabilities, then you have to know how to create a promise.

Imagine, for example, that we want to create a promise-based setup function that combines the *connection*, *channel*, and *queue* tasks from above. The function should return a promise, and the caller should be able to do something like this:

setup().then(onSetupSuccess).catch(onSetupFailed);

In this example, onSetupSuccess is a caller-defined callback function for the case that the entire setup (that is, the creation of the connection *and* channel *and* queue) succeeds, and onSetupFailed is a caller-defined callback function for the case that any of these tasks fail.

A promise can be created in the following way, as described [here](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#Creating_a_Promise) in the docs:

new Promise(executor);

In this case, executor is a reference to a so-called **executor function***.*You have to define this executor function when you create a promise.

The executor function must have the following signature:

function executor(resolve, reject)

That is, it must take two arguments, and both arguments are function references. Don’t worry, you **don’t** have to define these functions too. But you have to call these functions in the body of the executor function.

In particular, you have to call the resolve function when you want to “fulfil” the promise, that is, when the asynchronous tasks succeeded. And you have to call the reject function when you want to “reject” the promise, that is, when the asynchronous task failed.

Finally, to create a promise-enabled asynchronous function, this function hast to simply return an appropriately set-up promise.

Let’s see how this looks in practice. Below is a program that defines and uses a setup function as described above:

var amqp = require('amqplib');

setup()  
 .then(() => myLog("Setup succeeded"))  
 .catch(err => myLog("Setup failed: " + err));

function setup() {  
 return new Promise(setupExecutor);  
}

function setupExecutor(resolve, reject) {  
 amqp.connect("amqp://localhost")  
 .then(connection => {  
 return connection.createChannel()  
 })  
 .then(channel => {  
 return channel.assertQueue("my\_queue");  
 })  
 .then(queue => {  
 resolve();  
 })  
 .catch(err => {  
 reject(err);  
 });  
}

myLog("I'm here");

If we execute this program without a running RabbitMQ server, we get the following output:

$ node app.js  
16:50:10.447 I'm here  
16:50:10.463 Setup failed: Error: connect ECONNREFUSED 127.0.0.1:5672

And with a running RabbitMQ server:

$ node app.js  
16:51:25.815 I'm here  
16:51:25.890 Setup succeeded

You can check on [**http://localhost:15672/**](http://localhost:15672/)again, the connection, channel, and queue should be there. Everything with a single call to the promise-enabled setup function. We have successfully create a new asynchronous promise-based function that in turn comprises three interdependent asynchronous tasks.

As always, it is more common to define function reference arguments as **anonymous functions** rather than named functions. In this case, the program would look like this (only the setup function changes):

var amqp = require('amqplib');

setup()  
 .then(() => myLog("Setup succeeded"))  
 .catch(err => myLog("Setup failed: " + err));

function setup() {  
 return new Promise((resolve, reject) => {  
 amqp.connect("amqp://localhost")  
 .then(connection => {  
 return connection.createChannel()  
 })  
 .then(channel => {  
 return channel.assertQueue("my\_queue");  
 })  
 .then(queue => {  
 resolve();  
 })  
 .catch(err => {  
 reject(err);  
 });  
 });  
}

myLog("I'm here");

Note that here the implementation of the executor function for our promise uses asynchronous promise-based functions itself (the amqplib API functions).

However, nothing stops us to use, for example, callback-based asynchronous functions to implement the promise executor function. Just because we are defining a promise doesn’t mean that we also have to implement all asynchronous operations inside the promise with promises.

If we wanted, we could use the **callback-based API** functions of amqplib for implementing our promise executor function. In this case, the program would look like this:

var amqp = require('amqplib/callback\_api');

setup()  
 .then(() => myLog("Setup succeeded"))  
 .catch(err => myLog("Setup failed: " + err));

function setup() {  
 return new Promise((resolve, reject) => {  
 amqp.connect("amqp://localhost", (err, connection) => {  
 if (err)  
 reject(err);  
 else {  
 connection.createChannel((err, channel) => {  
 if (err)  
 reject(err);  
 else {  
 channel.assertQueue("my\_queue", null, (err, queue) => {  
 if (err)  
 reject(err);  
 else {  
 resolve();  
 }  
 });  
 }  
 });  
 }  
 });  
 });  
}

myLog("I'm here");

However, as stated above, I think it goes without saying that the promise-based version is more readable than the callback-based version.

## Appendix: myLog Function

function myLog(msg) {  
 console.log(getTime() + " " + msg);  
}

function getTime() {  
 var d = new Date();  
 return d.toLocaleTimeString('en-GB') + "." + d.getMilliseconds();  
}