NodeJS CallBack

What Is a Callback?

If you are familiar with using jQuery, you are probably using callbacks on a regular basis. A *callback* is a function that is passed as an argument to another function and is generally called when the first function is finished. In the following jQuery example, a paragraph tag is hidden using jQuery's hide() method. This method takes an optional argument of a callback function. If a callback function is given as an argument, it will be called when the hiding of the paragraph is complete. This makes it possible to do something when the hiding has finished—in this case, showing an alert.

```
$('p').hide('slow', function() {
alert("The paragraph is now hidden");
});
```

A callback can be optional, however. If you do not want a callback, you could write this code:

```
$('p').hide('slow');
```

Comparing the two code examples, the first one adds an anonymous function as a second argument and is called once the first function has finished. In the second example, there is no callback.

Because functions in JavaScript are first-class objects, they can be passed as arguments to other functions in this way. This allows you to write code that says, "Do this and when you are finished doing that, do this." To illustrate the difference between writing code with and without callbacks, you look at two jQuery examples in the browser.

How Node.js Uses Callbacks

Node.js uses callbacks everywhere and especially where any I/O (input/output) operation is happening. Consider the following example where Node.js is used to read the contents of a file from disc using the filesystem module:

```
var fs = require('fs');
fs.readFile('somefile.txt', 'utf8', function (err, data) {
if (err) throw err;
console.log(data);
});
```

Here's what's happening:

1. The `fs` (filesystem) module is required so it can be used in the script.
2. The `fs.readFile` method is given a path to a file on a filesystem as a first argument.
3. A second argument of `utf8` is given to indicate the encoding of the file.
4. A third argument of a callback function is given to the `fs.readFile` method.
5. The callback function takes a first argument of `err` that will hold any errors returned from reading the file.
6. The callback function takes a second argument of `data` that will hold the data returned

from reading the file.

7. Once the file has been read, the callback will be called.
8. If `err` is true, an error will be thrown.
9. If `err` is false, the data from the file is available and can be used.
10. In this case, the data is logged to the console.

You will see callbacks being used in this way over and over again in Node.js. Another example of this is the `http` module. The `http` module allows developers to create http clients and servers.
You have already seen the `http` module in use with the Hello World server. The `http.get()` method from the `http` module allows requests to be made to a web server and for the response data to be used somehow.

```
var http = require('http');
http.get({ host: 'shapeshed.com' }, function(res) {
console.log("Got response: " + res.statusCode);
}).on('error', function(e) {
console.log("Got error: " + e.message);
});
```

An explanation of this code is as follows:

1. The `http` module is required, so it can be used in the script.
2. The `http.get()` method is given two arguments.
3. The first is an object of the options. In this example, it is instructed to fetch the home page of shapeshed.com.
4. The second argument is the callback that takes the response as an argument.
5. When the response is returned from the remote server, the callback function is fired.
6. Within the callback function, the response status code is logged or, in the case of an error, this is logged.

The callback function is called when the response comes back from the remote server and not before. This example demonstrates much of what Node.js is trying to achieve.
As this piece of
code has to go out to a network (the Internet) to fetch data, it is not possible to know exactly when or even if the data will return.

Particularly when you are fetching data from multiple sources and multiple networks, writing code that responds to the unpredictable nature of when data will return can be difficult.

Node.js is a response to this problem and aims to provide a platform for creating networked applications. Callbacks are one of the key ways that Node.js approaches network programming because they allow code to be run when another event happens (in this case, data being returned from shapeshed.com). Callbacks are said to be “fired” when events happen that cause the callback function be called.

Demonstrating Network I/O and Callbacks

```
var fs = require('fs'),
http = require('http');
```

```
http.get({ host: 'shapeshed.com' }, function(res) {
console.log("Got a response from shapeshed.com");
}).on('error', function(e) {
console.log("There was an error from shapeshed.com");
});
fs.readFile('file1.txt', 'utf8', function (err, data) {
if (err) throw err;
console.log('File 1 read!');
});
http.get({ host: 'www.bbc.co.uk' }, function(res) {
console.log("Got a response from bbc.co.uk");
}).on('error', function(e) {
console.log("There was an error from bbc.co.uk");
});
fs.readFile('file2.txt', 'utf8', function (err, data) {
if (err) throw err;
console.log('File 2 read!');
});
```

When the code runs, it does the following:

1. Fetches the home page of shapeshed.com.
2. Reads the contents of file1.txt.
3. Fetches the home page of bbc.co.uk.
4. Reads the contents of file2.txt.

Looking at the example, can you tell which one will return first? A good guess would be that the two files that are being read from disk are likely to return first as they do not have to go out to the network.

After that, though, it is difficult to say which of the files being read will return first because you do not know how big the files are. As for fetching the two home pages, the script goes out to the network, and the response time depends on a number of things that are difficult to predict.

The Node.js process will also not exit while it has registered callbacks that have not yet fired. Callbacks are first a way to account for unpredictability, but also an efficient way to deal with concurrency (or doing more than one thing at once).

Synchronous and Asynchronous Code

Node.js runs on a single process and dictates that developers use an asynchronous style of coding.
In the last example, you saw how four operations were performed asynchronously through the callback pattern. Coding in an asynchronous way is not specific to Node.js or JavaScript, though. It is a style of programming.
Synchronous code means that operations are performed one at a time and that, until one operation is over, the code execution is blocked from moving onto the next operation

The Event Loop

At this point, you might be wondering how all of this magic happens. Node.js uses JavaScript’s event loop to support the asynchronous programming style that it advocates.

This can be another tricky concept to come to grips with, but it basically allows callback functions to be saved and then run at a point in the future when an event happens. This might be data being returned
from a database or an HTTP request returning data. Because the execution of the callback function is deferred until the event happens, there is no need to halt the execution, and control can be returned to the Node runtime environment so that other things can happen.

As you have seen with the blocking and non-blocking examples, using an event
loop is a different way of programming. Some developers refer to it as writing programs inside out, but the idea is that you structure your code around events rather than an expected order of inputs. Because the event loop is based on a single process, there are some rules that you should follow to ensure high performance:

- Functions must return quickly.
- Functions must not block.
- Long-running operations must be moved to separate processes.