Node.js

JavaScript based framework.
Easy Guide Book

Henry Rowland

Node.js

JavaScript based framework Easy Guide Book

By Rick L.

Table of Contents

Introduction

NodeJS is a very useful framework. It can assist you in the development of network applications which are highly scalable. It is also good for web development. If you are a JavaScript expert, then it will be easy for you to learn Node. This book has simplified the process of learning Node.JS.

Chapter 1- A Brief Overview

NodeJS was developed for the purpose of helping programmers in the development of network applications which are scalable. It is an event-driven and asynchronous framework which was built on top of the JavaScript engine for Chrome. For you to master how to use it well, you should be good in JavaScript. It is good for the development of web applications. Due to its non-blocking mode of operation, the programs are not forced to wait for the others to be executed, but a callback is just made. As you wait for a response in Node, you are allowed to continue with performing other tasks. All of the Node libraries have been designed so that they can operate in a non-blocking mode.

The data for a node application is not buffered at all in the memory. Instead, this data is output by means of chunk-by-chunk. Due to its features, it has demonstrated a great difference between it and other related frameworks and hence its increasing popularity.

Chapter 2- Management of Module Dependencies

Some applications are very dependent on the number of NPM modules that are to be used. These can then be specified in the file “*package.json*” as shown below:

```
“dependencies”: {
    “express”: “2.3.12”,
    “jade”: “>= 0.0.1”,
    “redis”:   “0.6.0”
}
```

Once you have done that, every time that you need your project to be fresh, the dependencies will have to be sorted out as shown below:

```
$ npm install
```

With this, one is allowed to either specify a specific version of the module or the minimum module that they need to use. This can be done by using the symbol “>=” so as to prefix the number.

Sometimes, you might have some dependencies which are related to development. You may not want to have these dependencies in the production, and these can be specified by the use of the property "*devDependencies.*" This is shown in the example given below:

```
"devDependencies": {
    "vows": ">= 0.4.x"
}
```

Our production by the use of "npm install –production" will make sure that the dependency is not installed. If you are working on a private module, the property "private": true" can be added to the file "*package.json*" so that the module is not accidentally published to the NPM registry.

For those who are in need of hosting their module in a private git repository, but who still have a need for it to be bundled as a dependency in the project, this can be done as follows:

```
"dependencies": {
  "secret-module": "git+ssh://git@github.com:username/secret-repo.git#v2.3"
}
```

The last part shown above having the URL has been used to specify the tag which is to be used. A branch name or a commit hash can also be specified.

Chapter 3- Require and Exports

Consider a NodeJS file having the content given below:

```
var p = 10;

var addP = function(value) {

    return value + p;

};
```

When we are in another file, it will be impossible for us to access the variable *"p"* or the function *"addP."* However, the reason behind this is not the use of the keyword *"var."* Note that a module is the fundamental building block of a file in NodeJS, and it is used for mapping to the file directly. In the above module, everything has been made private. However, when programming in Node, we don't want things to be accessible or visible only in the current file but from other files.

Before we can explore how a particular module can be exposed, it will be good for us to know how a particular module is loaded. The *"require"* function is used in this case. It is used for the purpose of loading a particular module, and the return value for this has to be assigned to a particular variable. This is shown below:

```
var myVar = require('./misc');
```

If our module is not intended to expose anything to the other modules or files, then the above will not be very important or useful. For the things to be exposed, we use “module.exports,” and anything that we need to expose will be exposed. This is shown in the code given below:

```
var p = 8;
var addP = function(value) {
    return value + p;
};
module.exports.p = p;
module.exports.addP = addP;
```

The module which has been loaded can then be used as shown below:

```
var myVar = require('./misc');
console.log("Addition of %d to 10 will gives us %d", myVar.p, myVar.addP(10));
```

Alternatively, the contents of a module can be exposed as shown below:

```
var MyUser = function(name, email) {
    this.name = name;
    this.email = email;
};
module.exports = MyUser;
```

As shown above, the difference may seem to be very minor, but at the same time very important. The exportation of the user has been done indirectly without having to perform any indirection. The difference between:

```
module.exports.MyUser = MyUser;
//v.s
```

```
module.exports = MyUser;
```

is just about the use as shown below:

```
var myuser = require('./user');

var user = new user.User();

//v.s

var user = new user();
```

It will not matter whether your module is just a container having exported modules or not. The two can be mixed within the module, but the resulting API may not be good at all.

You should also consider what is to happen in case the function is exported. This is shown in the code given below:

```
var pLevel = function(level) {

return level > 9000 ? "it is over 9000!!!" : level;

};

module.exports = pLevel;
```

Whenever the above function is required, the return value will be the actual function. That means that we can do the following:

```
require(‘./plevel’)(9050);
```

The above code is a condensed version of the following code:

```
var pLevel = require(‘./plevel’)

pLevel(9050);
```

Chapter 4- Asynchronous Flow Control using Promise

A promise is an object which represents the result of a function call which is asynchronous. In some communities, they are referred to as "*futures.*" In NodeJS, there are numerous CommonJS promises, and these will be explored in this chapter of the book.

They were once based on EventEmitters as part of the Node framework. This is what they looked like:

```
var promise = fs.stat("test");
promise.addListener("success", function (value) {
   // ok
})
promise.addListener("error", function (err) {
   //the error
});
```

With this style, more allocations were needed than was necessary. Political issues with these also arose since there was no fulfillment of all the contracts advocating the use of promises. Due to this, the people had to do away with this style of using promises, and then adopt a new way in which callbacks are used. These can be used as shown in the code given below:

```
fs.stat("test", function (err, value) {
    if (err) {
        // the error
    } else {
        // ok
    }
});
```

Terminologies

There are numerous terminologies which are associated with the use of promises. These are discussed below:

- Fulfillment: callbacks are called with the value once a successful promise has been fulfilled. If registration of more callbacks is done in the future, all will be called by use of the same value. Fulfillment means the process of asynchronous analog for returning a particular value.

- Rejection: when it is impossible for a promise to be fulfilled, the promise is “rejected,” and this invokes the errbacks which are waiting and remembers the error which was rejected for the future errbacks which are attached. It represents the asynchronous analog in which throwing an exception is done.

- Progressback: this is a function which has been executed so as to show that progress has been made towards the resolution of a promise.

- Callback: this is a function which has been executed once a promise has been fulfilled with a value.

- Resolution: this is a promise which has been resolved, and then it makes progress towards a fulfillment or a rejection. Resolving of a promise can only be done once, and this can be done with a promise instead of a fulfillment or a rejection.

- Errback: this is a function which has been executed if a promise has been rejected, with an exception.

The CommonJS promises for Node can be implemented in two main ways. These are discussed below:

Q Library

The NPM command can be used for installation of this library as shown below:

```
npm install q
```

It can then be used as follows in the nodeJS program:

```
var q = require(‘q’);
```

Promised-IO

Again, the NPM command can be used for installation of this as shown below:

```
npm install promised-io
```

According to or depending on the version that you are using, include it by use of the following command:

```
var q = require('promised-io/lib/promise'); // <=v2.3
var q = require('promised-io/promise'); // >=v2.4
```

Note that the two are compatible with each other, and are able to use the promises of each other, which means that you can use the one that you want, depending on your preference. However, they provide different helper functions outside of our basic functions which are responsible for the creation and consumption of promises.

On the client side, this should be as follows:

```
$.when($.get(…))
.then(function(value) {
   console.log(‘successful!’)
}, function(error) {
   console.log(‘ A rejection’);
});
```

The resolving of a promise is done only once. This means that it can either be resolved or rejected. It also means that an individual errback or callback can only be resolved once. Once a promise has been resolved with a value, it will always remember the fulfillment. Once a future callback has been attached to the promise, the value which was resolved previously will be used for its execution. The same case applies to errbacks. If a rejection of a callback occurs and then an errback is attached after that, the rejected value will then be used for its execution. The behavior of promises will be the same, regardless of whether they are resolved or they will be resolved in the future.

Thenables

This is another name for promises in commonJS. The name was derived from the methods which were available on the promise. The method was responsible for the purpose of attaching callbacks, errbacks, and progressbacks. This is shown in the example given below:

```
promise.then(function() {
  // callback will be executed on a successful promise resolution
}, function() {
  // errback will be executed on a rejection
}, function() {
  // progressback will be executed if the promise has a progress to report
});
```

When

This is a function which assists in attaching listeners to an object. In this case, one is not sure of whether the object is a promise or not. Methods are expected to return a direct value or a promise object, and this is the importance of using this function. In the case of a return of a promised value, then the "*then*" method will not be available. If the function "*then*" is not used in the promise object, the method "*when*" will take over so as to ensure that it behaves well by making sure that the calling of callbacks is done in separate events and not more than once. This is shown in the example given below:

```
when(myPromise, function() {
  // callback to be executed on a successful promise resolution or if myPromise is not a
promise but just a value
}, function() {
  // errback to be executed on a rejection
}, function() {
  // progressback to be executed if the promise has a progress to report
});
```

Bubbling

Once a value has been returned by a callback, it is bubbled up the chain of promises. The same case applies for rejections. With this, the handling of rejections can be done at a higher level than the function which had called our promise-returning function which has been rejected. Also, APIs which are promise-based will be able to compose responses for successes and rejections by changing of the error or the bubble which has been bubbled. Consider the example given below for ease of understanding:

```
function performAsync() {
  return asyncHelper().then(function(value) {
    // perform an extra processing on the value
    return value; <— becoming the resolution of the returned promise by the
performAsync function
  });
}
performAsync().then(function(value) {
  console.log(‘resolved’, value);
}, function(error) {
  // this will receive rejections from the performAsync or  be bubbled from the
asyncHelper
  console.log(‘error’, error);
});
```

Using Deferred Helper to create Promises

A Deferred is an object which can be used for creation and manipulation of promises. It has a promise property with that which it can use for referencing the property that it is managing. The promise is resolved or rejected by the use of the *“reject”* or *“resolve”* methods. A value or a promise can be passed to the *“resolve”* method. If a value is passed to this method, then the promise will be fulfilled. If a promise is passed, then the resolution of that promise will be forwarded to this. Note that resolving of a promise can only be done once.

Reject is just a method which will accept a reason for the rejection. In most cases, the rejection is of a string type, but it can be of any type. In the case of Deferred, the promise is rejected with a rejection reason. Consider the code given below whih shows how this can be done:

```
function performAsync() {
  var deferred = q.defer();
  setTimeout(function() {
    deferred.resolve(‘hello there’);
  }, 500);
  return deferred.promise;
}
performAsync().then(function(value) {
  console.log(‘Promise has been Resolved!’, value);
});
```

Most of the functions which take the callback style of Node can easily be wrapped in a promise. Any callback function which takes the Node style, and that which calls its callback only once, is a potential candidate for wrapping. Consider the following function which demonstrates how this can be done:

```
function myFunction(nodeAsyncFn, context) {
  return function() {
    var defer = q.defer()
      , args = Array.prototype.slice.call(arguments);
    args.push(function(error, value) {
      if (error !== null) {
        return defer.reject(error);
      }
      return defer.resolve(value);
    });
    nodeAsyncFn.apply(context || {}, args);
    return defer.promise;
  };
};
```

The above function can be used together with other node functions such as the “*fs.readFile*”. as it is shown below:

```
var rFile = promisify(fs.readFile);
readFile(‘file.txt’).then(function(d) {
  console.log(d);
});
```

With both PromisedIO and Q, one can provide utilities which can be used for calling or wrapping Node-style functions. This is shown below:

```
var rFile = q.convertNodeAsyncFunction(fs.readFile);
readFile(‘file.txt’)
.then(function (d) { });
// or
q.execute(fs.readFile, ‘file.txt’)
.then(function (d) { });
```

The above case shows how it can be done by use of *“PromisedIO.”* When using Q, it can be done as follows:

```
var rFile = q.node(fs.readFile);
readFile(‘file.txt’)
.then(function (d) { });
// or
q.ncall(fs.readFile, fs, ‘file.txt’)
.then(function (d) { });
// or
var deferred = q.defer();
fs.readFile(‘file.txt’, deferred.node());
return deferred.promise;
```

Chapter 5- Management of Node.js callback Hell

Consider the code given below:

```
doAsync1(function () {
doAsync2(function () {
doAsync3(function () {
doAsync4(function () {
})
})
})
```

That is what the callback hell is. When dealing with callback hell, you have to know that it is subjective. The reason for this is that when using code which is heavily nested, it can be the one working finely. With asynchronous code, when the management of the flow becomes overly complex, then the matter becomes hellish. For you to know the hell you are undergoing, consider the amount of refactoring that you are required to do.

We want to demonstrate this by use of an example. Consider a situation in which we are interested in finding the largest file contained in a particular directory. The module given below can be used for doing this:

```
var fLargest = require('./findLargest')
fLargest('./path/to/dir', function (error, filename) {
if (error) return console.error(error)
console.log('The largest file is:', filename)
})
```

The following steps can be followed so as to achieve the above needed work:

- Reading all of the files available in the provided directory.

- Get the statics of each file available in the directory.

- Determine the largest file amongst the available ones (if there are multiple of them with the same size, just pick one).

- Use the name of the largest file to callback.

In case an error results at any point in the process, the callback should be done by the use of that error. Again, the callback should not be called more than once.

A Nested Approach

First, we should use the nested approach to this. An example of how this can be done is shown below:

```
var fs = require('fs')

var path = require('path')

module.exports = function (direct, callb) {

fs.readdir(direct, function (error, files) { // [1]

if (error) return callb(error)

var counter = files.length

var errored = false

var statistics = []

files.forEach(function (file, index) {

fs.statistics(path.join(direct,file), function (error, stat) { // [2]

if (errored) return

if (error) {

errored = true

return callb(error)
}
statistics[index] = stat // [3]
if (—counter == 0) { // [4]
var largest = statistics
.filter(function (statistic) { return statistic.isFile() }) // [5]
.reduce(function (prev, next) { // [6]
if (prev.size > next.size) return prev
return next
})
cb(null, files[statistis.indexOf(largest)]) // [7]
}
})
})
})
}
```

Note that you should check for the presence of parallel operations. If you find any, then you are responsible for making sure that they complete. Only regular files should be grabbed, but links and directories should be left untouched. Often, we have to reduce the size of the largest files that we have so as to get the largest one of all. The final file should then be pulled with the associated callback and statistics. The only problem in implementation of this is on how to deal with the parallel operations and ensure that the callback is made only once. However, this will be handled later.

Consider the code given below:

```
function getStatistics (paths, callb) {
var counter = paths.length
var errored = false
var statistics = []
paths.forEach(function (path, index) {
fs.statistic(path, function (error, statistic) {
if (errored) return
if (error) {
errored = true
return callb(error)
}
statistics[index] = statistic
if (—counter == 0) callb(null, statistics)
})
})

}
```

In the above code, we have begun by grabbing our files from the directory. Our first task is the “***fs.readdir(),”*** and this means that we will not be required to write a function for that. With the above code, we will provide a set of paths, and the function will then return the statistics of those paths and their order will be maintained.

It is now time for us to implement the part for comparison. We need to be able to compare the different files available in our directory so as to determine the largest one of all. This should return the largest file in the directory. The code for doing that should be implemented as shown below:

```
function getLargestFile (files, statistics) {

var largest = statistics

.filter(function (statistic) { return statistic.isFile() })

.reduce(function (prev, next) {

if (prev.size > next.size) return prev

return next
})

return files[statistics.indexOf(largest)]

}
```

That is how it should be done. The whole thing can then be tied together as shown below:

```
var fs = require(‘fs’)

var path = require(‘path’)

module.exports = function (directory, callb) {

fs.readdir(direct, function (error, files) {

if (error) return callb(error)

var paths = files.map(function (file) { // [1]

return path.join(diretory,file)

})

getStats(paths, function (error, statistics) {

if (error) return callb(error)

var lFile = getLargestFile(files, stats)

cb(null, lFile)

})
})

}
```

That is how the code should be. A list of paths should then be generated from the directory and the files. With the modular approach, testing and reusing of methods is made much easier. The export itself can easily be reasoned about. At this point, the parallel statistics are easily managed.

Async Approach

This module is widely used, and it will take the same way taken by node in performing operations. The async module can be used as shown below:

```
var fs = require(‘fs’)

var async = require(‘async’)

var path = require(‘path’)

module.exports = function (diretory, callb) {

async.waterfall([ // [1]

function (next) {

fs.readdir(directory, next)

},

function (files, next) {

var paths =

files.map(function (file) { return path.join(directory,file) })

async.map(paths, fs.stat, function (error, statistics) { // [2]

next(error, files, statistics)

})
},

function (files, statistics, next) {

var largest = statistics

.filter(function (statistic) { return statistic.isFile() })

.reduce(function (prev, next) {

if (prev.size > next.size) return prev

return next

})

next(null, files[statistics.indexOf(largest)])

}

], callb) // [3]

}
```

With the property “*async.waterfall*,” a series of flow of execution will be provided, you will be in a position to pass data from a particular operation to another function, and this will be done by the use of the “*next*” callback.

With the function “*async.map*,” we will be in a position to run the “*fs.stat*” in parallel, and these will call back with an array having the results. You have also spotted the function “*callb.*” This will be called in case an error occurs along the way, or once the last step has been completed. The calling will be done only once.

With the async module, there will be a guarantee that only one of the callbacks will be fired. It is also good in propagation of errors and management of parallelism on our behalf.

A promises approach

With promises in NodeJS, one is provided with functional programming and error handling perks. I know you are wondering how that can achieve any use of promises. Let us use the Q module so as to show how this can be implemented. This is shown below:

```
var fs = require(‘fs’)

var path = require(‘path’)

var Q = require(‘q’)

var fs_readdirectory = Q.denodeify(fs.readdir) // [1]

var fs_statistic = Q.denodeify(fs.statistic)

module.exports = function (directory) {

return fs_readdir(directory)

.then(function (files) {

var promises = files.map(function (file) {

return fs_stat(path.join(directory,file))

})

return Q.all(promises).then(function (statistics) { // [2]

return [files, statistics] // [3]

})
})

.then(function (data) { // [4]

var files = data[0]

var statistics = data[1]

var largest = statistics

.filter(function (statistic) { return statistic.isFile() })

.reduce(function (prev, next) {

if (prev.size > next.size) return prev

return next

})

return files[statistics.indexOf(largest)]

})

}
```

Note that the core functionality of Node is not aware of promise, but we can make it to be so. The statistics alls will be run in parallel by the "*Q.all,*" and the order of the resulting array will be maintained. Our aim is to pass our files and the statistics to our "*next*" function, and this will be the last thing to be returned. Note that any of the exceptions which will be thrown inside the promise object will be caught, unlike what we have had in our previous examples. The API for the client will also be modified so that it becomes promise centric. This is shown below:

```
var fLargest = require(‘./findLargest’)

fLargest(‘./path/to/directory’)

.then(function (fname) {

console.log(‘The largest file is:’, fname)

})

.catch(console.error)
```

From the design shown above, you don’t have to expose a promise interface in this case. With most promise libraries, a mechanism is provided on which we can expose a style for the nodeback.

A generators approach

Generators are just lightweight co-routines which can be used in JavaScript. With these, one can use the *"yield"* keyword so as to suspend and then resume a particular function. The functions for generators make use of a special syntax which looks like "***function* ().***" Asynchronous operations can also be suspended and resumed by use of objects such as the thunks or promises, and this can lead to asynchronous code which looks as if it is synchronous. A thunk is just a function which will return a callback other than calling it. The callback in this case will have a similar signature to the typical nodeback function, meaning that the error will form the first argument.

We need to create an example whivh will demonstrate how generators can be used for the purpose of asynchronous flow control. This is how it can be done:

```
var c = require('co')

var thunk = require('thunkify')

var fs = require('fs')

var path = require('path')
var readdirectory = thunk(fs.readdirectory) <strong>[1]</strong>

var statistic = thunk(fs.statistic)
```

```
module.exports = c(function* (directory) { // [2]

var files = yield readdiretory(directory) // [3]

var statistics = yield files.map(function (myfile) { // [4]

return statistic(path.join(diretory,file))

})

var largest = statistics

.filter(function (statistic) { return statistic.isFile() })

.reduce(function (prev, next) {

if (prev.size > next.size) return prev

return next

})

return files[statistics.indexOf(largest)] // [5]

})
```

The Node core functionality is not aware of thunk, but we should make it be so. With the variable “*c,*” we will take a generator function in which we will be in a position to use the “*yield*” keyword so as to suspend it. The generator function will be suspended until the “*readdiretory*” has been returned. The value which results from this will then be returned from the “*files*” variable. The variable “*c*” can also handle arrays which a set of parallel operations will perform. The resulting array whose order is maintained will be assigned to the variable “*statistics.*” It is after this that the final result will be returned.

This generator function can also be consumed using the same callback API which we had specified at the beginning of this chapter. With the "*c,*" appropriate error handling mechanisms will be implemented and any resulting errors including exceptions raised will be passed to our callback function. Around the statement for yield, generators allow us to use the common "*try/catch*" block or statement for the purpose of handling errors and exceptions. This is shown in the example given below:

```
try {
var files = yield readdirectory(directory)
} catch (error) {
console.error('something went wrong while reading the directory')

}
```

Co provides us with a mechanism on how arrays, nested generators, objects, promises, and others can be handled. Other generator modules are also coming as they are under implementation. With the Q module, there is a method named "*Q.async*" which will behave in a similar to the co which uses generators.

Wrapping up

There are methods which can help you get control over how your application flows. The generator idea is one of the available ways how this can be done. With the modular approach, one can apply it to whichever flow libraries that they are using. Examples of flow libraries include the async, promises, and generators.

Chapter 6- Promise Methods

With most promise objects, we usually have the static counterparts in the main objet which is “Q,” and this usually accepts either a promise or a non-promise object and a fulfilled promise will then be created first. With all others, you will have static counterparts which will be named similarly to the promise object.

With some methods, the used names should be the same as the keywords which are reserved in JavaScript. Examples of these include the *“try,” “catch,”* and *“finally.”* With this, a very clear parallel between the asynchronous promise operations and the standard synchronous language constructs will be shown.

Main promise methods

<u>promise.then(onFulfilled, onRejected, onProgress)</u>

This is just the "*then*" method having some additional progress handler.

Callbacks to promises

Consider the asynchronous Node callback shown below:

```
readFile(function (error, data) {
if (error) return console.error(error)
console.log(data)
})
```

In case the function "readFile" has returned to us a promise, then we could have done this:

```
var p = readFile()

p.then(console.log, console.error)
```

When you look at it shallowly, you will see that we have changed only the aesthetic. However, at this moment, we have a value which can be used for representation of the asynchronous operation which is the promise. The promise can then be passed around, and any person having access to the promise will be able to consume it by use of "*then,*" independent of whether or not the asynchronous operation has completed. We are somehow guaranteed that our asynchronous operation will not change since the promise will be resolved for only once, meaning that it can either be fulfilled or rejected.

The "*then*" should be seen as a function which can be used for unwrapping a promise so as to reveal what had taken place in the asynchronous operation. Avoid seeing it as a function which takes only two callbacks which include "*onFulfilled*" and "*onRejected.*" One can unwrap it by use of "*then,*" provided you have access into it.

Nesting and Chaining Promises

The method *"then"* can be used for the purpose of returning a promise. This is shown below:

```
var p1 = readFile()

var p2 = p1.then(readAnotherFile, console.error)
```

The promise just represents the return value for its own *"onRejected"* and *"onFulfilled"* handlers in case they had been specified. Note that it is possible for us to have only one resolution, and the promise will proxy any resolution which has been called as shown below:

```
var p1 = readFile()

var p2 = p1.then(function (data) {

return readAnotherFile() // if readFile runs successfully, let us readAnotherFile

}, function (error) {

console.error(error) // if readFile runs unsuccessfully, let us log it but we will still
readAnotherFile
return readAnotherFile()
})
p2.then(console.log, console.error) // the product of readAnotherFile
```

Note that *“then”* returns a promise, and this indicates that the promises can be chained for the purpose of avoiding the callback hell. This is shown below:

```
readFile()
.then(readOtherFile)
.then(performSomethingElse)
.then(…)
```

If there is a need for you to keep alive a closure, the promises can be nested as shown below:

```
readFile()
.then(function (d) {
return readOtherFile().then(function () {
// perform something with the `data`
})

})
```

Synchronous Functions and Promises

There are a number of ways that promises can be used for the modeling of synchronous functions. An example of this is when we use the "*return*" function so as to call another function. In our previous examples, we have been using the function "*readAnotherFile()*" so as to signal what comes next after the function "*readFile*."

Once you have returned a promise, the next "*then*" will be signaled once the synchronous operation has been completed. Any value of another type can also be returned, and the next "*onFulfilled*" will have a value as an argument passed to it. This is shown below:

```
readFile()

.then(function (buffer) {

return JSON.parse(buffer.toString())

})

.then(function (data) {

// doing something with the `data`

})
```

That is how it can be done.

Promises and Error Handling

Other than the “*return*” keyword, one can use the other keywords such as “*throw*” and “*try/catch,*” which are very useful when it comes to error and exception handling. This shows how powerful promises are in Node. We want to synchronize how this can be done. Consider a situation in which we have the following piece of synchronous code:

```
try {
   doThis()
   doThat()
} catch (error) {
   console.error(error)
}
```

With the above example, the functions “*doThis()*” and “*doThat()*” will return errors. Our aim is to catch and then log this error. With the “*try/catch*” block, so many operations can be grouped. This means that the process of handling errors explicitly for each of the operation we have can be greatly avoided. This can be done asynchronously with the use of promises. This is shown below:

```
doThisAsync()
.then(doOtherAsync)
.then(null, console.error)
```

In case the function *“doThisAsync”* runs unsuccessfully, the promise will be rejected and the next then in the chain which has the handler for *“onRejected”* will be called. In our example, this is the function “*console.error.*” Note that just as it is the case with the block “*try/catch,*” this function will never be called. That is how much we have proved since with the raw callbacks, we were handling the errors explicitly at each step. When an exception has been thrown, whether it is either implicit or explicit, it will be handled by the promise if it is from the “*then*” callback. An example of this is shown below:

```
doThisAsync()
.then(function (d) {
   d.test.baz = ‘bar’ // throwing a ReferenceError as test is not defined

})
.then(null, console.error)
```

An error has resulted. The *"ReferenceError"* which is raised in the above case will be handled by the "*onRejected*" handler which is next in the chain. The above can also work for an explicit flow.

An example of this is shown below:

```
doThisAsync()
.then(function (d) {
    if (!d.baz) throw new Error('A baz was expected to be there')
})
.then(null, console.error)
```

As you are aware, promises work in the same way as "*try/catch*" block does. With this block, one can mask an error without having to handle it explicitly. An example of this is shown below:

```
try {
    throw new Error('never will realize this has happened')
} catch (ex) {}
```

The same case applies to promises as shown below:

```
readFile()
.then(function (d) {
    throw new Error('never will realize this has happened')
})
```

For us to expose the errors which have been masked, we should use the clause ".then(null, onRejected)" so as to end the promise chain. An example of this is shown below:

```
readFile()
.then(function (d) {
    throw new Error('now I realize this had happened')
})
.then(null, console.error)
```

There are libraries which have other mechanisms for how masked errors can be exposed. A good example is the *“Q”* module, which provides us with the method named *“done”* which can be used for rethrowing the error upstream.

Conversion of callbacks to promises

You may not know how promises are generated. The specification of the API for the creation of a promise is not done in the “*Promise/A+*” since this is not necessary for interoperability. This is an indication that the implementation of promise libraries always vary.

Note that with the Node core asynchronous functions, promises cannot be returned. However, by use of the Q module, we can make this one possible. This can be done as shown below:

```
var fs_rFile = Q.denodeify(fs.readFile)

var p = fs_rFile(‘file.txt’)

p.then(console.log, console.error)
```

With Q, a number of helper functions are provided for adaptation of Node and other environments so that they can be aware of promises.

Creation of Raw Promises

You can use "*Q.defer*" for the purpose of the creation of raw promises. Suppose that we want to make "fs.readFile" be promise-aware by wrapping it. This can be done as shown below:

```
function fs_rFile (file, encoding) {
    var deferred = Q.defer()
fs.readFile(file, encoding, function (error, data) {
if (error) deferred.reject(error) // rejecting the promise with `error` as the reason

else deferred.resolve(d) // fulfilling the promise with the `d` as the value

})
return deferred.promise // the promise will be returned
}
fs_rFile('file.txt').then(console.log, console.error)
```

One can also make APIs which can support both APIs and callbacks. You are now aware of how a callback code can be turned into a promise code. APIs can be developed which can be used for provision of an interface for both callback and promise. An example is when we want to turn the “fs.readFile” into an API which can support both promises and callbacks. This can easily be done. The code given below demonstrates how this can be done in Node:

```
function fs_rFile (file, encoding, callback) {
var deferred = Q.defer()
fs.readFile(function (error, d) {
if (error) deferred.reject(error) // rejecting the promise with the `error` as the reason

else deferred.resolve(d) // fulfilling the promise with the`d` as the value


})
return deferred.promise.nodeify(callback) // the promise will be returned
}
```

Once a callback has been provided, the standard Node arguments style is used for calling it once the promise has been resolved or rejected. The standard Node style arguments are of the form (err, result). An example of this is shown below:

```
fs_rFile(‘file.txt’, ‘utf8’, function (error, data) {
// …
})
```

Promises can also be used for the purpose of performing parallel operations. You are now aware of how sequential asynchronous operations operate in Node.JS. When using the Q module so as to deal with parallel operations, we use the method “*Q.all*.” This method usually takes in an array of promises, but then it returns a new promise which it creates. Once all of the necessary operations have been completed in a successful manner, then the new promise will be completed. In case single or multiple operations fail or run unsuccessfully, then the new promise will be rejected. An example of this is shown below:

```
var aPromise = Q.all([ fs_rFile(‘myfile1.txt’), fs_readFile(‘myfile2.txt’) ])

aPromise.then(console.log, console.error)
```

That is how it can be done. You should also be aware that the functionality of promises mimics that of functions. A function always has a single return value. With promises, there is much consistency with the asynchronous parts. For the purpose of spreading results into multiple arguments, one can use the “*Q.spread*.”

Chapter 7- Execution of Multiple Instances in a Single Process

There is a frequent request for Node to be integrated with other applications. An example is when we need to integrate it with the other event loops and support multiple execution contexts for the Node. When we talk of the same context, we mean the ability to have multiple instances of Node existing peacefully in the same application.

Currently, with Node v0.12, multiple execution contexts can be used within the same event loop. The Node internals should also be made aware of the presence of multiple execution contexts. Consider the example given below, showing how this can be done:

```
function myconnect() {
this.write('GET / HTTP/1.1\r\n' +
'Host: strongloop.com\r\n' +
'\r\n');
this.pipe(process.stdout);
}
require('net').connect(80, 'strongloop.com', myconnect);
```

When looked at execution-wise, it will be as follows:

<enter VM>

connect(80, ‘strongloop.com’); // kernel reporting EINPROGRESS here

<leave VM>

<wait for the TCP handshake to complete>

<enter VM>

myconnect();

<leave vm>

The execution context can change between our calls to the vm. This is why the function “*connect()*” has to remember the current context and the call to “*myconnect()*” should restore it. If you fail to do this, you might find it really hard for you to track any available bugs in your program.

However, with Node, this is automatically taken care of. This means that you don’t have to worry about it.

For those who need an add-on author and you need to make the addon, be aware of context, it can be done as follows:

*** Instead of NODE_MODULE(), use the NODE_MODULE_CONTEXT_AWARE(). Before:**

```
void Initialize(v8::Handle<v8::Object> exports) {

// …

}

NODE_MODULE(test, Initialize)
```

For after, do it as follows:

```
void Initialize(v8::Handle<v8::Object> exports,

v8::Handle<v8::Value> module,

v8::Handle<v8::Context> context) {

// …

}

NODE_MODULE_CONTEXT_AWARE(test, Initialize);
```

The context-aware initialize will be called once for each context which has been created by the embedder. There exists no per-context cleanup context. The function "*node::AtExit()*" is currently a per-process event, but we need it to fill up the above role. The global variables can also be turned into per-context properties.

Chapter 8- LoopBack Node.js Framework

The LoopBack is just an open-source project which can welcome contributions.

Loopback-component-explorer

The loopback-explorer was renamed to “loopback-component-explorer.” The reason behind this is that it is now a full-fledged loopback component. The version 2.0.0 of the module was released, which has two breaking changes.

Since it is a loopback component, it can be loaded and configured via the “*server/component-config.json*” as shown below:

```
// server/component-config.json
{
“loopback-component-explorer”: {
“mountPath”: “/explorer”
}
```

}

For the migration from the 1.x version range to be simplified, a middleware should be provided, and this should be mountable in any Express application in the same way. An example of this is shown below:

```
var explorer = require('loopback-component-explorer');
// v1.x - will not work anymore
application.use('/explorer', explorer(applicTION, options));
// v2.x
application.use('/explorer', explorer.routes(application, options));
```

Note that the use of “*component-config.json*” so as to load components is a new feature. Its documentation is expected to be released very soon.

Loopback core

With all the user models and the methods for applications, the promise-based invocation can be supported,   as well as the callback-based calls. Methods which are related to replication such as the “*PersistedModel.replicate*” can also support promises.

A filter which is method-based can also be specified for the middleware. Consider the example given below, which shows some specified limits for the middleware to get   only the requests. Here is the code for the example:

```
// server/middleware.json
{
“auth”: {
“my-middleware”: {
“methods”: [“GET”],
“params”: [“argument1”, “argument2”],
“enabled”: true
}
}
}
```

A wrapper has also been implemented around the middleware errorhandler for express. The wrapper will add a configuration option for excluding error stack traces from the HTTP responses. Consider the typical configuration shown below for disabling the stack traces when it is being executed in a production environment. Here is the example:

```
// server/middleware.json
{
“final:after”: {
“loopback#errorHandler”: {}
}
}
// server/middleware.production.json
{
“final:after”: {
“loopback#errorHandler”: {
“params”: {
“includeStack”: false
}
}
}
}
```

The loopback-datasource-juggler

A support for assisting when querying embedded models was added together with the array values for MongoDB whenever the memory connector is being used. Consider the example given below, which shows the new querying capabilities introduced in Node:

```
// common/models/file.json

{

“properties”: {

“name”: “string”,

“sons”: [“string”],

“friends”: [{ “name”: “string” }]

}

}
```

If you are in need of finding the people who have a son named “*John,*” then the following query can be used:

File.find({ where: { son: “John” } })

That is how the query can be written. Suppose you are in need of finding the people who have a friend with the name “*Joel.*” The query can be formulated as shown below:

File.find({ where: { “friends.name”: “Joel” } })

The support for the query “*RegExp*” was also introduced and supported for use in queries.

Consider a situation in which you want a person whose name starts with a particular letter, such as “*N.*” The query for this can be formulated as shown below:

File.find({ where: { regexp: /^N/ } })

That is how the query should be formulated. If more than one person has a name starting with the letter “*N,*” then all of them will be shown in the output or the result after executing the query.

Other than the memory connector, there are some other connectors which are supported by the operator “*RegExp.*” These include the following:

- PostgreSQL
- MongoDB
- MySQL
- MS SQL

Lastly, in all the model-discovery methods, support for promises was added.

The loopback-connector-mongodb

The flag “*allowExtendedOperators*” was recently implemented. When this flag has been enabled, the data for update operations will include the MongoDB operators such as the “*$rename.*”

There are two ways how this flag can be enabled.

First, if the file for the model definition has the property “settings.mongodb.allowExtendedOperators” having been set to “*true*” as shown in the example given below:

```
{
“name”: “MyModel”,
“settings”: {
“mongodb”: {
“allowExtendedOperators”: true
```

Second, when the update method is being called from the code, the setting of the flag can be done in the options object as shown below:

User.updateAll(

{ name: ‘Joel’ },

{ ‘$rename’: { name: ‘fname’ }},

{ allowExtendedOperators: true });

Chapter 9- Event Loop Monitoring

With the event loop in Node, we are given the capability of handling highly concurrent requests while at the same time running the requests which are single-threaded. The processing is delegated to the operating system itself, and this is done via the POSIXD interface as the new callbacks and events are still being handled.

Blocking an event loop

When this is done in a Node application, it can lead to catastrophic effects. We will demonstrate how the blocking can be done by external events.

With a function call, blocking is said to have occurred when the flow of the current execution of the thread waits until we have the function finished before being executed. Most people see the “I/O” as blocking. An example is when you are calling the function “socket.read().” The program will have to wait for our call to end so as to continue. This is what you may need to perform with a return value.

Note that in Node, a function calling a CPU work will always block. Consider the fibonacci function shown below, which will block the execution of the thread as per current since it needs to use the CPU. Here is the function:

```
function fibonacci (j) {

if (j < 2)

return 1;

else

return fibonacci(j-2) + fibonacci(j-1);

}
```

In Node, we do not create servers for Fibonacci. However, sometimes, the Node becomes CPU bound. Consider the example given below:

```
function requestHandler(request, response) {

var body = request.rawBody; // has the POST body

try {

var json = JSON.parse(body);

response.end(json.user.username);

}

catch(ex) {

response.end(“FAIL”);
}

}
```

With the above example, the body of the request will be taken and then parsed on. The flow of this should run very well until one has executed a file which is worth 15MB. When the call "JSON.parse()" has been executed on 15MB file, the processing will take around 1.5 seconds.

For those who are in need of monitoring concurrent connections, you can use the "*StrongLoop Arc*." It will also enable you to monitor the throughput of your Node applications that you have created.

Conclusion

It can be concluded that NodeJS is a very useful and very powerful library which provides an assist in the development of web applications. The framework is well known for its apps, which are highly scalable in terms of size and performance. This is why it is highly used in the development of network applications in which high scalability is needed.

The framework is event-driven and asynchronous, and was built on top of the JavaScript engine for Chrome. JavaScript developers and experts usually find it easy for them to learn how to use the Node since they are loosely related. In fact, Node is based on JavaScript since it is one of its available libraries.

In Node, your programs will not be forced to wait for certain operations to be completed. As you wait for the response to be available to you, you can continue in performing other tasks. This is because Node is a non-blocking JavaScript blocking and can support concurrent operations to be executed. Most of you are aware of the concept of buffering of data in which it is stored in a type of memory named buffer for quick and easy access. In Node, this type of memory or the process is not supported at all. What happens is that the data is provided or output to the user in the form of chunk-by-chunk.

The framework has great features, which explains the reason behind its increased popularity among web developers. When compared to related frameworks or the ones which can supplement it, it is more popular. An example of such a framework is the Ruby on rails. The framework has numerous modules which can be used for the purpose of accomplishing different tasks in one's app. It will be good for you to learn how to use this framework so as to get assistance in your process of development. This book is an excellent guide for you.