**Node.js Promise Tutorial**

[**https://www.guru99.com/node-js-promise-generator-event.html**](https://www.guru99.com/node-js-promise-generator-event.html)

In previous tutorials, you would have seen callback functions which are used for Asynchronous events. But sometimes callback functions can become a nightmare when they start becoming nested, and the program starts to become long and complex.

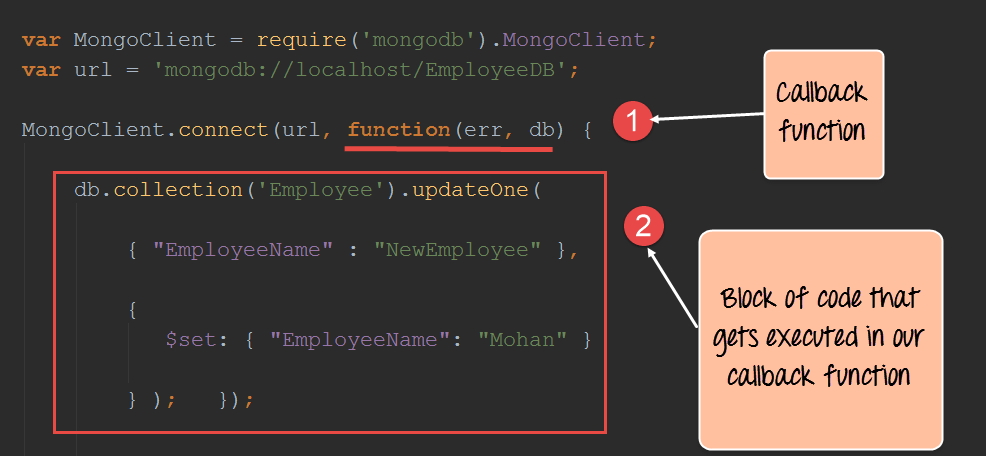
In this tutorial, you will learn-

* [What are promises](https://www.guru99.com/node-js-promise-generator-event.html#1)
* [Callbacks to promises](https://www.guru99.com/node-js-promise-generator-event.html#2)
* [Dealing with nested promises](https://www.guru99.com/node-js-promise-generator-event.html#3)
* [Creating a custom promise](https://www.guru99.com/node-js-promise-generator-event.html#5)

**What are promises**

Before we start with promises, let's first revisit what are "callback" functions in Node.js. We have seen these callback functions a lot in the previous chapters, so let's quickly go through one of them.

The example below shows a code snippet, which is used to connect to a[MongoDB](https://www.guru99.com/mongodb-tutorials.html)database and perform an update operation on one of the records in the database.

[](https://www.guru99.com/images/NodeJS/010716_0659_NodejsTutor1.png)

1. In the above code, the part of the function(err,db) is known as the declaration of an anonymous or callback function. When the MongoClient creates a connection to the MongoDB database, it will return to the callback function once the connection operation is completed. So in a sense, the connection operations happens in the background, and when it is done, it calls our callback function. Remember that this is one of the key points of Node.js to allow many operations to happen concurrently and thus not block any user from performing an operation.
2. The second code block is what gets executed when the callback function is actually called. The callback function just updates one record in our MongoDB database.

So what is a promise then? Well, a promise is just an enhancement to callback functions in Node.js. During the development lifecycle, there may be an instance where you would need to nest multiple callback functions together. This can get kind of messy and difficult to maintain at a certain point in time. In short, a promise is an enhancement to callbacks that looks towards alleviating these problems.

The basic syntax of a promise is shown below;

var promise = doSomethingAync()

promise.then(onFulfilled, onRejected)

* "doSomethingAync" is any callback or asynchronous function which does some sort of processing.
* This time, when defining the callback, there is a value which is returned called a "promise."
* When a promise is returned, it can have 2 outputs. This is defined by the 'then clause'. Either the operation can be a success which is denoted by the 'onFulfilled' parameter. Or it can have an error which is denoted by the 'onRejected' parameter.

**Note:** So the key aspect of a promise is the return value. There is no concept of a return value when working with normal callbacks in Node.js. Because of the return value, we have more control of how the callback function can be defined.

In the next topic, we will see an example of promises and how they benefit from callbacks.

**Callbacks to promises**

Now let's look at an example of how we can use "promises" from within a Node.js application. In order to use promises in a Node.js application, the 'promise' module must first be downloaded and installed.

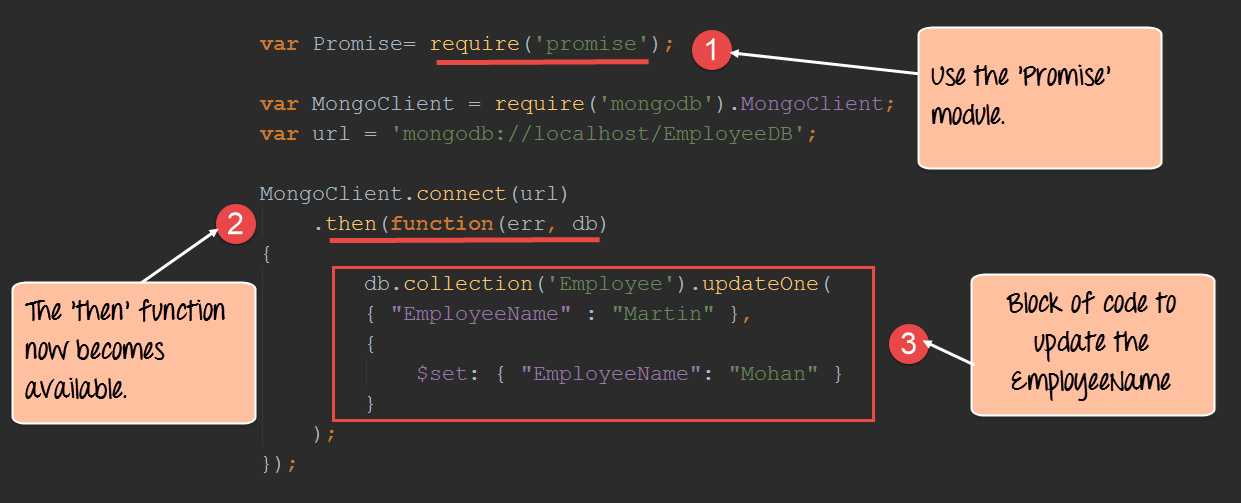
We will then modify our code as shown below, which updates an Employeename in the 'Employee' collection by using promises.

**Step 1)** Installing the NPM Modules

To use Promises from within a Node JS application, the promise module is required. To install the promise module, run the below command

**npm install promise**

**Step 2)**Modify the code to include promises

[](https://www.guru99.com/images/NodeJS/010716_0659_NodejsTutor2.png)

var Promise = require('promise');

var MongoClient = require('mongodb').MongoClient;

var url = 'mongodb://localhost/EmployeeDB';

MongoClient.connect(url)

.then(function(err, db) {

db.collection('Employee').updateOne({

"EmployeeName": "Martin"

}, {

$set: {

"EmployeeName": "Mohan"

}

});

});

**Code Explanation:-**

1. The first part is to include the 'promise' module which will allow us to use the promise functionality in our code.
2. We can now append the 'then' function to our MongoClient.connect function. So what this does is that when the connection is established to the database, we need to execute the code snippet defined thereafter.
3. Finally, we define our code snippet which does the work of updating EmployeeName of the employee with the name of "Martin" to "Mohan".

**Note:-**

If you now check the contents of your MongoDB database, you will find that if a record with EmployeeName of "Martin" exists, it will be updated to "Mohan."

To check that the data has been properly inserted in the database, you need to execute the following commands in MongoDB

1. Use EmployeeDB
2. db.Employee.find({EmployeeName :Mohan })

The first statement ensures that you are connected to the EmployeeDb database. The second statement searches for the record which has the employee name of "Mohan".

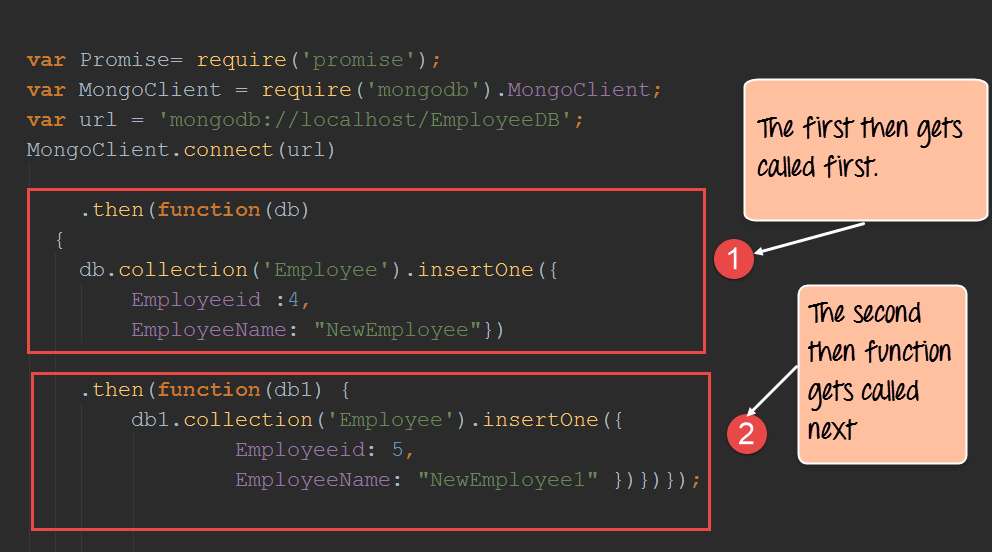
**Dealing with nested promises**

When defining promises, it needs to be noted that the "then" method itself returns a promise. So in a sense, promises can be nested or chained to each other.

In the example below, we use chaining to define 2 callback functions, both of which insert a record into the MongoDB database.

(**Note**: Chaining is a concept used to link execution of methods to one another. Suppose if your application had 2 methods called 'methodA' and 'methodB.' And the logic was such that 'methodB' should be called after 'methodA,' then you would chain the execution in such a way that 'methodB' gets called directly after 'methodA.')

The key thing to note in this example is that the code becomes cleaner, readable and maintainable by using nested promises.

[](https://www.guru99.com/images/NodeJS/010716_0659_NodejsTutor3.png)

var Promise = require('promise');

var MongoClient = require('mongodb').MongoClient;

var url = 'mongodb://localhost/EmployeeDB';

MongoClient.connect(url)

.then(function(db) {

db.collection('Employee').insertOne({

Employeeid: 4,

EmployeeName: "NewEmployee"

})

.then(function(db1) {

db1.collection('Employee').insertOne({

Employeeid: 5,

EmployeeName: "NewEmployee1"

})

})

});

**Code Explanation:-**

1. We are now defining 2 "then" clauses which get executed one after the other. In the first then clause, we are passing the 'db' parameter which contains our database connection. We are then using the collection property of the 'db' connection to insert records into the 'Employee' collection. The 'insertOne' method is used to insert the actual document into the Employee collection.
2. We are then using the 2nd then clause also to insert another record into the database.

If you now check the contents of your MongoDB database, you will find the 2 record's inserted into the MongoDB database.

**Creating a custom promise**

A custom promise can be created by using a node module called 'q.' The 'q' library needs to be downloaded and installed using the node package manager. After using the 'q' library, the method "denodeify" can be called which will cause any function to become a function which returns a promise.

In the example below, we will create a simple function called "Add" which will add 2 numbers. We will convert this function into a function to return a promise.

Once that is done, we will use the promise returned by the Add function to display a message in the console.log.

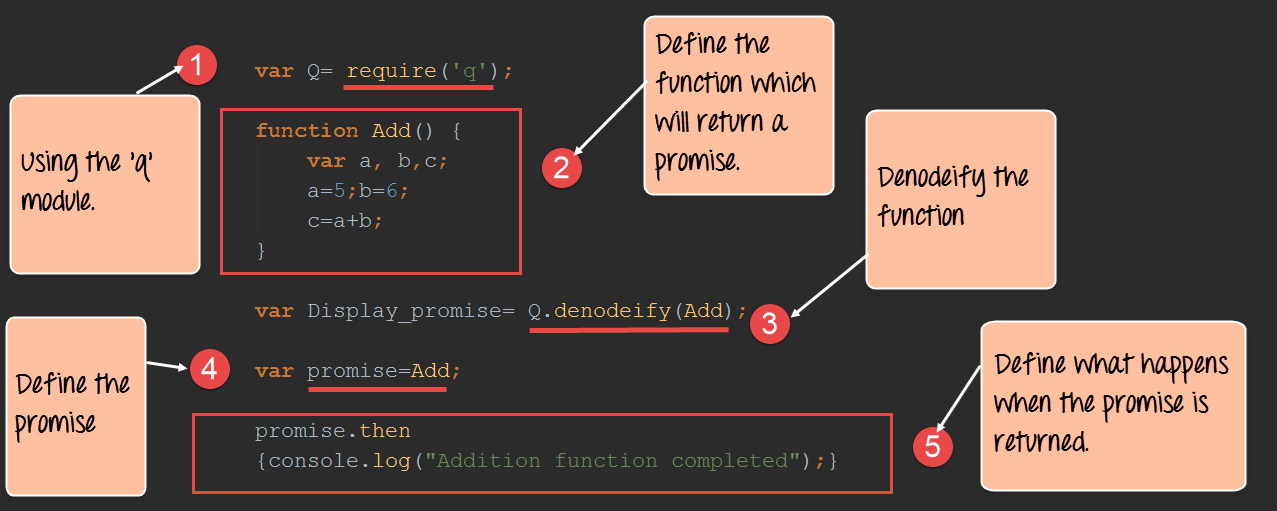
Let's follow the below steps to creating our custom function to return a promise.

**Step 1)** Installing the NPM Modules

To use 'q' from within a Node JS application, the 'q' module is required. To install the 'q' module, run the below command

**npm install q**

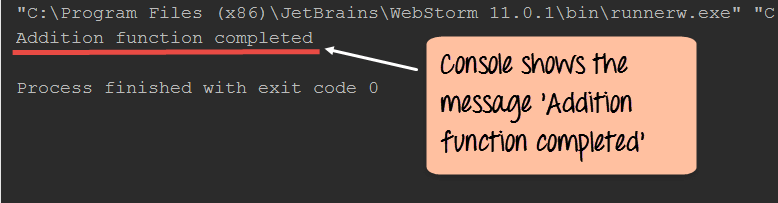
**Step 2)** Define the following code which will be used to create the custom promise.

[](https://www.guru99.com/images/NodeJS/010716_0659_NodejsTutor7.png)

**Code Explanation:-**

1. The first bit is to include the 'q' library by using the require keyword. By using this library, we will be able to define any function to return a callback.
2. We are creating a function called Add which will add 2 numbers defined in variables a and b. The sum of these values will be stored in variable c.
3. We are then using the q library to denodeify ( the method used to convert any function into a function that would return a promise) our Add function or in otherwise convert our Add function to a function which returns a promise.
4. We now call our "Add" function and are able to get a return promise value because of the prior step we performed of denodeify the Add function.
5. The 'then' keyword is used specify that if the function is executed successfully then display the string "Addition function completed" in the console.log.

When the above code is run, the output "Addition function completed" will be displayed in the console.log as shown below.

[](https://www.guru99.com/images/NodeJS/010716_0659_NodejsTutor8.png)

**Summary**

* Using callback functions in Node.js does have its disadvantages. Sometimes during the process of development, the nested use of callback functions can make the code messier and difficult to maintain.
* Most of the issues with nested callback functions can be mitigated with the use of promises and generators in node.js
* A Promise is a value returned by an asynchronous function to indicate the completion of the processing carried out by the asynchronous function.
* Promises can be nested within each other to make code look better and easier to maintain when many asynchronous function need to be called at the same time.

**Promises**

## [Aren't promises just callbacks?](https://stackoverflow.com/questions/22539815/arent-promises-just-callbacks)

Promises are not callbacks. A promise represents the **future result of an asynchronous operation**. Of course, writing them the way you do, you get little benefit. But if you write them the way they are meant to be used, you can write asynchronous code in a way that resembles synchronous code and is much more easy to follow:

api().then(function(result){

return api2();

}).then(function(result2){

return api3();

}).then(function(result3){

// do work

});

Certainly, not much less code, but much more readable.

But this is not the end. Let's discover the true benefits: What if you wanted to check for any error in any of the steps? It would be hell to do it with callbacks, but with promises, is a piece of cake:

api().then(function(result){

return api2();

}).then(function(result2){

return api3();

}).then(function(result3){

// do work

}).catch(function(error) {

//handle any error that may occur before this point

});

Pretty much the same as a try { ... } catch block.

Even better:

api().then(function(result){

return api2();

}).then(function(result2){

return api3();

}).then(function(result3){

// do work

}).catch(function(error) {

//handle any error that may occur before this point

}).then(function() {

//do something whether there was an error or not

//like hiding an spinner if you were performing an AJAX request.

});

And even better: What if those 3 calls to api, api2, api3 could run simultaneously (e.g. if they were AJAX calls) but you needed to wait for the three? Without promises, you should have to create some sort of counter. With promises, using the ES6 notation, is another piece of cake and pretty neat:

Promise.all([api(), api2(), api3()]).then(function(result) {

//do work. result is an array contains the values of the three fulfilled promises.

}).catch(function(error) {

//handle the error. At least one of the promises rejected.

**Bluebird Promises**

<https://www.guru99.com/bluebird-promises.html>

**There is some mistakes in the above link documentation, the code has some changes. The working code I have written here in this doc.**

Bluebird is a fully-featured Promise library for JavaScript. The strongest feature of Bluebird is that it allows you to "promisify" other Node modules in order to use them asynchronously. Promisify is a concept applied to callback functions. This concept is used to ensure that every callback function which is called returns some sort of value.

So if a Node JS module contains a callback function which does not return a value, if we Promisify the node module, all the function's in that specific node module would automatically be modified to ensure that it returns a value.

So you can use BlueBird to make the[MongoDB](https://www.guru99.com/mongodb-tutorials.html)module run asynchronously. This just adds another level of ease when writing Node.js applications.

We will look at an example of how to use the bluebird module.

Our example will first establish a connection to the "Employee collection" in the "EmployeeDB" database. If "then" connection is established, then it will get all of the records in the collection and display them in the console accordingly.

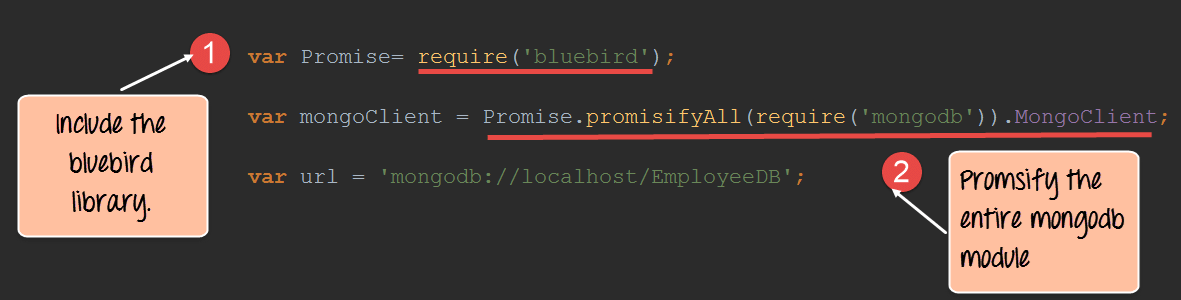
**Generating promises with the BlueBird library**

**Step 1)** Installing the NPM Modules

To use Bluebird from within a Node application, the Bluebird module is required. To install the Bluebird module, run the below command

**npm install bluebird**

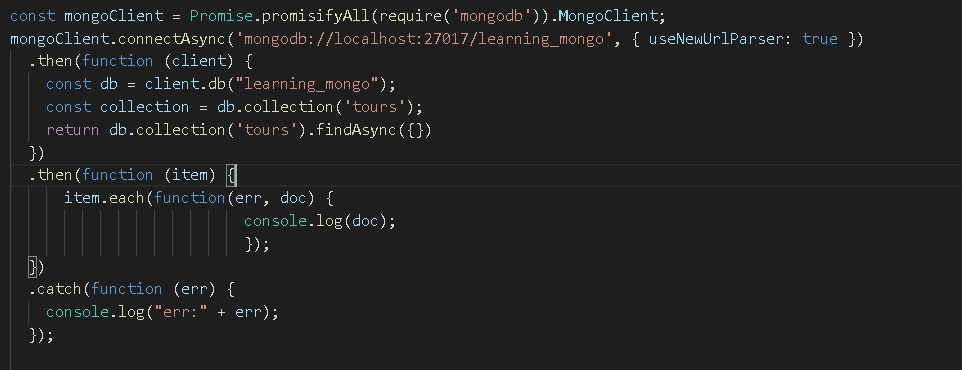
**Step 2)**The next step is to include the bluebird module in your code and promisify the entire MongoDB module. By promisify, we mean that bluebird will ensure that each and every method defined in the MongoDB library returns a promise.

[](https://www.guru99.com/images/NodeJS/010716_0659_NodejsTutor4.png)

**Code Explanation:-**

1. The require command is used to include the Bluebird library.
2. Use Bluebird's .promisifyAll() method to create an async version of every method the MongoDB module provides. This ensures that each method of the MongoDB module will run in the background and ensure that a promise is returned for each method call in the MongoDB library.

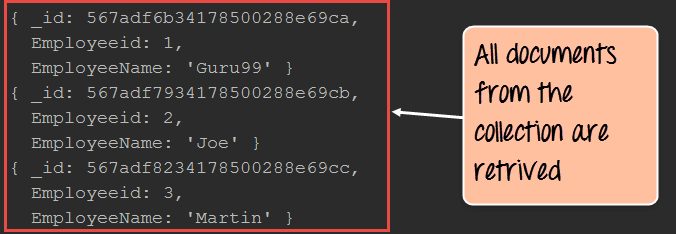
**Step 3)** The final step is to connect to our database, retrieve all the records in our collection and display them in our console log.



**Code Explanation:-**

1. You will notice that we are using the "connectAsync" method instead of the normal connection method for connecting to the database. Bluebird actually adds the Async keyword to each method in the MongoDB library to distinguish those calls which return promises and those which don't. So there is no guarantee that methods without the Async word will return a value.
2. Similar to the connectAsync method, we are now using the findAsync method to return all of the records in the mongoDB 'Employee' collection.
3. Finally, if the findAsync returns a successful promise we then define a block of code to iterate through each record in the collection and display them in the console log.

If the above steps are carried out properly, all of the documents in the Employee collection will be displayed in the console as shown in the output below.

[](https://www.guru99.com/images/NodeJS/010716_0659_NodejsTutor6.png)

Here is the code for your reference

var Promise = require('bluebird');

const mongoClient = Promise.promisifyAll(require('mongodb')).MongoClient;

mongoClient.connectAsync('mongodb://localhost:27017/learning\_mongo', { useNewUrlParser: true })

.then(function (client) {

const db = client.db("learning\_mongo");

const collection = db.collection('tours');

return db.collection('tours').findAsync({})

})

.then(function (item) {

item.each(function(err, doc) {

console.log(doc);

});

})

.catch(function (err) {

console.log("err:" + err);

});