

Node.js MongoDB Tutorial with Examples

15-20 minutos

Mostly all modern day web applications have some sort of data storage system at the backend to store data. For example, if you take the case of a web shopping application, data such as the price of an item or the number of items of a particular type would be stored in the database.

The Node js framework has the ability to work with databases which are commonly required by most modern day web applications. Node js can work with both relational (such as Oracle and MS SQL Server) and non-relational databases (such as MongoDB). During this tutorial, we will see how we can use databases from within Node js applications.

In this tutorial, you will learn-

- [Node.js and NoSQL Databases](#)
- [Using MongoDB and Node.js](#)
- [How to build a node express app with MongoDB to store and serve content](#)

Node.js and NoSQL Databases

Over the years, NoSQL database such as [MongoDB](#) and MySQL have become quite popular as databases for storing data. The ability of these databases to store any sort of content and particularly in any sort of format is what makes these databases so famous.

Node.js has the ability to work with both MySQL and MongoDB as databases. In order to use either of these databases, you need

to download and use the required modules using the Node package manager.

For MySQL, the required module is called "mysql" and for using MongoDB the required module to be installed is "Mongoose."

With these modules, you can perform the following operations in Node.js

1. Manage the connection pooling – Here is where you can specify the number of MySQL database connections that should be maintained and saved by Node.js.
2. Create and close a connection to a database. In either case, you can provide a callback function which can be called whenever the "create" and "close" connection methods are executed.
3. Queries can be executed to get data from respective databases to retrieve data.
4. Data manipulation such as inserting data, deleting and updating data can also be achieved with these modules.

For the remaining topics, we will look at how we can work with MongoDB databases within Node.js.

Using MongoDB and Node.js

As discussed in the earlier topic, MongoDB is one of the most popular databases used along with Node.js.

During this chapter, we will see

How we can establish connections with a MongoDB database

How we can perform the normal operations of reading data from a database as well as inserting, deleting and updating records in a mongoDB database.

For the purpose of this chapter, let's assume that we have the below mongoDB data in place.



Database name: EmployeeDB

Collection name: Employee

Documents

```
{
  {Employeeid : 1, Employee Name :
Guru99},
  {Employeeid : 2, Employee Name : Joe},
  {Employeeid : 3, Employee Name :
Martin},
}
```

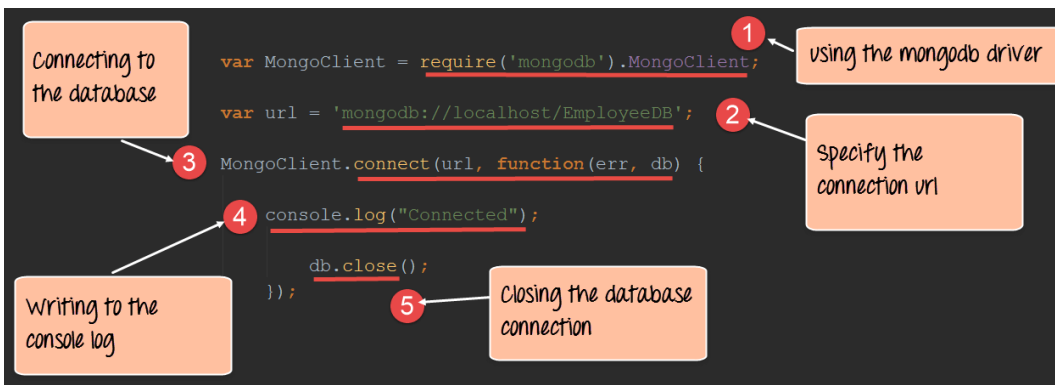
1. Installing the NPM Modules

To access Mongo from within a Node application, a driver is required. There are number of Mongo drivers available, but MongoDB is among the most popular. To install the MongoDB module, run the below command

npm install mongodb

2. Creating and closing a connection to a MongoDB database.

The below code snippet shows how to create and close a connection to a MongoDB database.



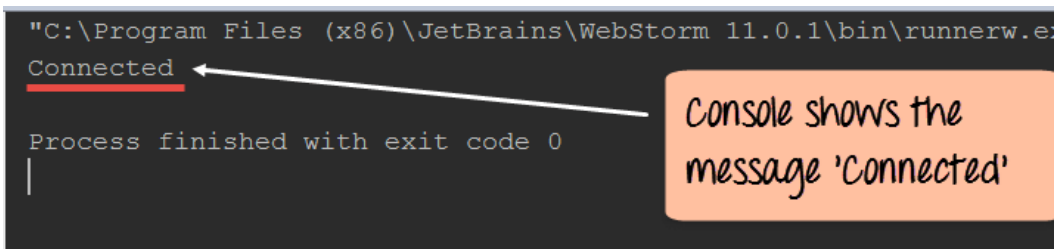
Code Explanation:

1. The first step is to include the mongoose module which is done through the require function. Once this module is in place, we can use the necessary functions available in this module to create connections to the database.
2. Next we specify our connect string to the database. In the

connect string there are 3 key values which are passed.

- The first is 'mongodb' which specifies that we are connecting to a MongoDB database.
 - The next is 'localhost' which means we are connecting to a database on the local machine.
 - The next is 'EmployeeDB' which is the name of the database defined in our MongoDB database.
3. The next step is to actually connect to our database. The connect function takes in our URL and has the facility to specify a callback function. It will be called when the connection is opened to the database. This gives us the opportunity to know if the database connection was successful or not.
 4. In the function, we are writing the string "Connection established" to the console to indicate that a successful connection was created.
 5. Finally, we are closing the connection using the db.close statement.

If the above code is executed properly, the string "Connected" will be written to the console as shown below.



```
"C:\Program Files (x86)\JetBrains\WebStorm 11.0.1\bin\runnerw.e
Connected
Process finished with exit code 0
|
```

Console shows the message 'Connected'

3. Querying for data in a MongoDB database – Using the MongoDB driver we can also fetch data from the MongoDB database.

The below section will show how we can use the driver to fetch all of the documents from our Employee collection (This is the collection in our MongoDB database which contains all the employee related documents. Each document has an object id, Employee name and employee id to define the values of the document) in our EmployeeDB

database.

The diagram shows a code snippet on a dark background with three orange callout boxes explaining different parts of the code. The code is as follows:

```
var MongoClient = require('mongodb').MongoClient;
var url = 'mongodb://localhost/EmployeeDB';

MongoClient.connect(url, function(err, db) {

  1 var cursor =db.collection('Employee').find( );

  cursor.each(function(err, doc) { 2
    3 console.log(doc);
  });
});
```

Callout boxes:

- Box 1 (top left): "Using the find function to create a cursor of records" with an arrow pointing to line 1.
- Box 2 (top right): "For each record in the cursor we are calling a function" with an arrow pointing to line 2.
- Box 3 (bottom left): "Printing the results to the console" with an arrow pointing to line 3.

```
var MongoClient =
require('mongodb').MongoClient;
var url = 'mongodb://localhost/EmployeeDB';

MongoClient.connect(url, function(err, db) {

    var cursor =
db.collection('Employee').find();

    cursor.each(function(err, doc) {

        console.log(doc);

    });
});
```

Code Explanation:

1. In the first step, we are creating a cursor (A cursor is a pointer which is used to point to the various records fetched from a database. The cursor is then used to iterate through the different records in the database. Here we are defining a variable name called cursor which will be used to store the pointer to the records fetched from the database.) which points to the records which are fetched from the MongoDB collection. We also have the facility of specifying the collection 'Employee' from which to fetch the records. The find() function is used to specify that we want to retrieve all of the documents from the MongoDB collection.

2. We are now iterating through our cursor and for each document in the cursor we are going to execute a function.
3. Our function is simply going to print the contents of each document to the console.

Note: - It is also possible to fetch a particular record from a database. This can be done by specifying the search condition in the find() function. For example, suppose if you just wanted to fetch the record which has the employee name as Guru99 then this statement can be written as follows "var cursor=db.collection('Employee').find()."

If the above code is executed successfully, the following output will be displayed in your console.

Output:

```
{ _id: 567adf6b34178500288e69ca,
  Employeeid: 1,
  EmployeeName: 'Guru99' }
{ _id: 567adf7934178500288e69cb,
  Employeeid: 2,
  EmployeeName: 'Joe' }
{ _id: 567adf8234178500288e69cc,
  Employeeid: 3,
  EmployeeName: 'Martin' }
```

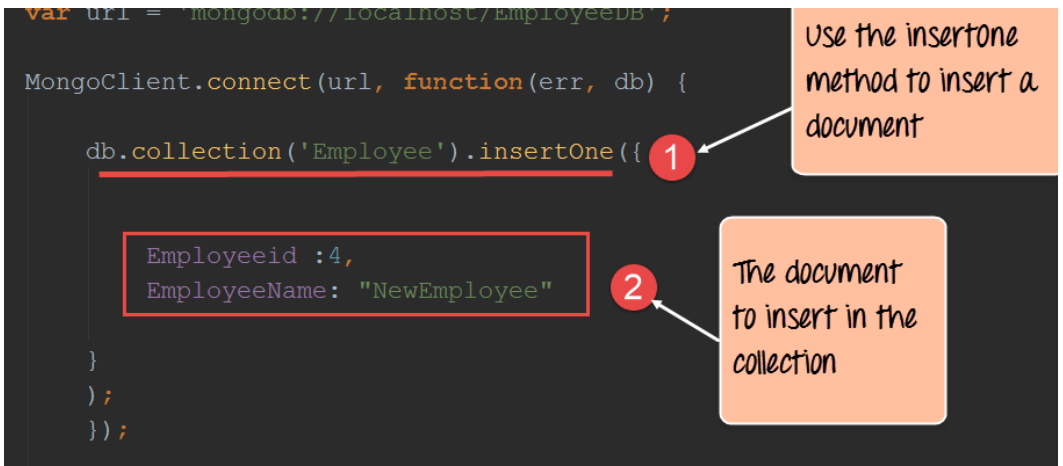
All documents from the collection are retrived

From the output,

- You will be able to clearly see that all the documents from the collection are retrieved. This is possible by using the find() method of the mongoDB connection (db) and iterating through all of the documents using the cursor.

4. **Inserting documents in a collection** – Documents can be inserted into a collection using the insertOne method provided by the MongoDB library. The below code snippet shows how we can insert a document into a mongoDB collection.

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



```
var MongoClient =
require('mongodb').MongoClient;
var url = 'mongodb://localhost/EmployeeDB';

MongoClient.connect(url, function(err, db) {

  db.collection('Employee').insertOne({
    Employeeid: 4,
    EmployeeName: "NewEmployee"
  });
});
```

Code Explanation:

1. Here we are using the insertOne method from the MongoDB library to insert a document into the Employee collection.
2. We are specifying the document details of what needs to be inserted into the Employee collection.

If you now check the contents of your MongoDB database, you will find the record with Employeeid of 4 and EmployeeName of "NewEmployee" inserted into the Employee collection.

Note: The console will not show any output because the record is being inserted in the database and no output can be shown here.

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({Employeeid :4 })`

The first statement ensures that you are connected to the EmployeeDb database. The second statement searches for the record which has the employee id of 4.

5. **Updating documents in a collection** - Documents can be updated in a collection using the `updateOne` method provided by the MongoDB library. The below code snippet shows how to update a document in a mongoDB collection.



```
var MongoClient =
require('mongodb').MongoClient;
var url = 'mongodb://localhost/EmployeeDB';

MongoClient.connect(url, function(err, db) {

  db.collection('Employee').updateOne({
    "EmployeeName": "NewEmployee"
  }, {
    $set: {
      "EmployeeName": "Mohan"
    }
  });
});
```

Code Explanation:

1. Here we are using the "updateOne" method from the MongoDB library, which is used to update a document in a mongoDB collection.

2. We are specifying the search criteria of which document needs to be updated. In our case, we want to find the document which has the EmployeeName of "NewEmployee."
3. We then want to set the value of the EmployeeName of the document from "NewEmployee" to "Mohan".

If you now check the contents of your MongoDB database, you will find the record with Employeeid of 4 and EmployeeName of "Mohan" updated in the Employee collection.

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

1. Use EmployeeDB
2. `db.Employee.find({Employeeid :4 })`

The first statement ensures that you are connected to the EmployeeDb database. The second statement searches for the record which has the employee id of 4.

6. **Deleting documents in a collection** - Documents can be deleted in a collection using the "deleteOne" method provided by the MongoDB library. The below code snippet shows how to delete a document in a mongoDB collection.

```
var MongoClient = require('mongodb').MongoClient;
var url = 'mongodb://localhost/EmployeeDB';

MongoClient.connect(url, function(err, db) {

  db.collection('Employee').deleteOne(
    { "EmployeeName" : "Mohan" }
  );
});
```



Use the deleteOne method

Search criteria for which record needs to be deleted

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

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

MongoClient.connect(url, function(err, db) {

    db.collection('Employee').deleteOne(

        {
            "EmployeeName": "Mohan"
        }

    );
});
```

Code Explanation:

1. Here we are using the "deleteOne" method from the MongoDB library, which is used to delete a document in a mongoDB collection.
2. We are specifying the search criteria of which document needs to be deleted. In our case, we want to find the document which has the EmployeeName of "Mohan" and delete this document.

If you now check the contents of your MongoDB database, you will find the record with Employeeid of 4 and EmployeeName of "Mohan" deleted from the Employee collection.

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

1. Use EmployeeDB
2. db.Employee.find()

The first statement ensures that you are connected to the EmployeeDb database. The second statement searches and display all of the records in the employee collection. Here you can see if the record has been deleted or not.

How to build a node express app with MongoDB to store and serve content

Building an application with a combination of both using express and MongoDB is quite common nowadays.

When working with [JavaScript](#) web based applications, one will normally hear of the term MEAN stack.

- The term MEAN stack refers to a collection of JavaScript based technologies used to develop web applications.
- MEAN is an acronym for MongoDB, ExpressJS, [AngularJS](#) and Node.js.

Hence, it's always good to understand how Node.js and MongoDB work together to deliver applications which interact with backend databases.

Let's look at a simple example of how we can use "express" and "MongoDB" together. Our example will make use of the same Employee collection in the MongoDB EmployeeDB database.

We will now incorporate Express to display the data on our web page when it is requested by the user. When our application runs on Node.js, one might need to browse to the URL

<http://localhost:3000/Employeeid>.

When the page is launched, all the employee id in the Employee collection will be displayed. So let's see the code snippet in sections which will allow us to achieve this.

Step 1) Define all the libraries which need to be used in our application, which in our case is both the MongoDB and express library.

```
var express= require('express');
var app=express();
var MongoClient = require('mongodb').MongoClient;
var url = 'mongodb://localhost/EmployeeDB';
var str="";
```

The code snippet is annotated with four numbered steps and callouts:

- 1**: *using the express module* (points to `require('express')`)
- 2**: *using our Mongodb module* (points to `require('mongodb')`)
- 3**: *creating the connection string* (points to `'mongodb://localhost/EmployeeDB'`)
- 4**: *creating a variable which will be used in our application* (points to `var str=""`)

Code Explanation:

1. We are defining our 'express' library, which will be used in our application.
2. We are defining our 'express' library, which will be used in our application for connecting to our MongoDB database.
3. Here we are defining the URL of our database to connect to.
4. Finally, we are defining a string which will be used to store our collection of employee id which need to be displayed in the browser later on.

Step 2) In this step, we are now going to get all of the records in our 'Employee' collection and work with them accordingly.

```
app.route('/Employeeid').get(function(req,res)
{
    MongoClient.connect(url,function(err,db) {
        var cursor = db.collection('Employee').find( ) ;
        cursor.each(function(err,item) {
            if(item != null) {
                str = str + "    Employee id    "
                    + item.Employeeid + "<br>";
            }
        });
    });
}
```

1 Create a route for our application

2 Get all the records in the Employee collection

3 Iterate through each record

4 Put all of the information in the 'str' variable

Code Explanation:

1. We are creating a route to our application called 'Employeeid.' So whenever anybody browses to **`http://localhost:3000/Employeeid`** of our application, the code snippet defined for this route will be executed.
2. Here we are getting all of the records in our 'Employee' collection through the `db.collection('Employee').find()` command. We are then assigning this collection to a variable called cursor. Using this cursor variable, we will be able to browse through all of the records of the collection.
3. We are now using the `cursor.each()` function to navigate through all of the records of our collection. For each record, we are going to define a code snippet on what to do when each record is accessed.

4. Finally, we see that if the record returned is not null, then we are taking the employee via the command "item.Employeeid". The rest of the code is just to construct a proper HTML code which will allow our results to be displayed properly in the browser.

Step 3) In this step, we are going to send our output to the web page and make our application listen on a particular port.

```
res.send(str);  
}); }  
  
var server=app.listen(3000,function()  
{})
```

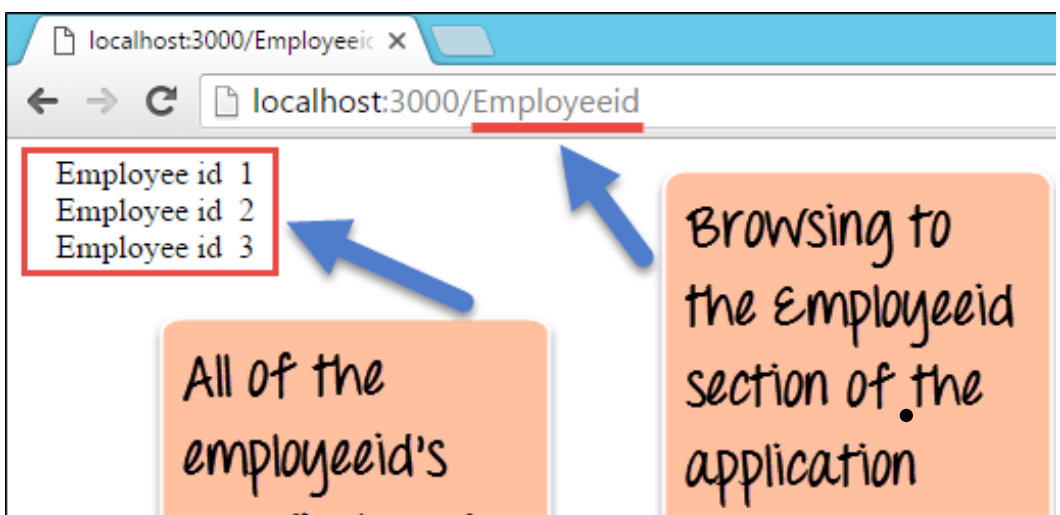
1 Send the content of the 'str' variable to the web page

2 Make our server application listen on port 3000

Code Explanation:

1. Here we are sending the entire content which was constructed in the earlier step to our web page. The 'res' parameter allows us to send content to our web page as a response.
2. We are making our entire Node.js application listen on port 3000.

Output:



are displayed

From the output,

- It clearly shows that all of the employeeid's in the Employee collection were retrieved. This is because we use the MongoDB driver to connect to the database and retrieve all the Employee records and subsequently used "express" to display the records.

Here is the code for your reference

```
var express = require('express');
var app = express();
var MongoClient =
require('mongodb').MongoClient;
var url = 'mongodb://localhost/EmployeeDB';
var str = "";

app.route('/Employeeid').get(function(req, res)
{
    MongoClient.connect(url, function(err,
db) {
        var cursor =
db.collection('Employee').find();
        //noinspection JSDeprecatedSymbols
        cursor.each(function(err, item) {

            if (item != null) {
                str = str + "    Employee
id " + item.Employeeid + "</br>";
            }
        });
        res.send(str);
        db.close();
    });
});

var server = app.listen(3000, function() {});
```

Note: cursor.each maybe deprecated based on version of your

MongoDB driver. You can append `//noinspection JSDeprecatedSymbols` before `cursor.each` to circumvent the issue. Alternatively, you can use `forEach`. Below is the sample code using `forEach`

```
var express = require('express');
var app = express();
var MongoClient =
require('mongodb').MongoClient;
var url = 'mongodb://localhost/EmployeeDB';
var str = "";

app.route('/Employeeid').get(function(req, res)
{
    MongoClient.connect(url, function(err, db) {
        var collection =
db.collection('Employee');
        var cursor = collection.find({});
        str = "";
        cursor.forEach(function(item) {
            if (item != null) {
                str = str + "    Employee id
" + item.Employeeid + "</br>";
            }
        }, function(err) {
            res.send(str);
            db.close();
        })
    });
});
var server = app.listen(8080, function() {});
```

Summary

- Node.js is used in conjunction with NoSQL databases to build a lot of modern days web applications. Some of the common databases used are MySQL and MongoDB.
- One of the common modules used for working with MongoDB databases is a module called 'MongoDB.' This module is installed via the Node package manager.
- With the MongoDB module, it's possible to query for records

in a collection and perform the normal update, delete and insert operations.

- Finally, one of the modern practices is to use the express framework along with MongoDB to deliver modern day applications. The Express framework can make use of the data returned by the MongoDB driver and display the data to the user in the web page accordingly.