# NODE JS

Node.js is an open source server environment.

Node.js allows you to run JavaScript on the server.

Learning by Examples

Our "Show Node.js" tool makes it easy to learn Node.js, it shows both the code and the result.

Example

var http = require('http');  
  
http.createServer(function (req, res) {  
  res.writeHead(200, {'Content-Type': 'text/plain'});  
  res.end('Hello World!');  
}).listen(8080);

**Click on the "Run example" button to see how it works.**

Examples Running in the Command Line Interface

In this tutorial there will be some examples that are better explained by displaying the result in the command line interface.

When this happens, The "Show Node.js" tool will show the result in a black screen on the right:

Example

console.log('This example is different!');  
console.log('The result is displayed in the Command Line Interface');

**Click on the "Run example" button to see how it works.**

What is Node.js?

* Node.js is an open source server environment
* Node.js is free
* Node.js runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.)
* Node.js uses JavaScript on the server

Why Node.js?

**Node.js uses asynchronous programming!**

A common task for a web server can be to open a file on the server and return the content to the client.

Here is how PHP or ASP handles a file request:

1. Sends the task to the computer's file system.
2. Waits while the file system opens and reads the file.
3. Returns the content to the client.
4. Ready to handle the next request.

Here is how Node.js handles a file request:

1. Sends the task to the computer's file system.
2. Ready to handle the next request.
3. When the file system has opened and read the file, the server returns the content to the client.

Node.js eliminates the waiting, and simply continues with the next request.

Node.js runs single-threaded, non-blocking, asynchronous programming, which is very memory efficient.

What Can Node.js Do?

* Node.js can generate dynamic page content
* Node.js can create, open, read, write, delete, and close files on the server
* Node.js can collect form data
* Node.js can add, delete, modify data in your database

What is a Node.js File?

* Node.js files contain tasks that will be executed on certain events
* A typical event is someone trying to access a port on the server
* Node.js files must be initiated on the server before having any effect
* Node.js files have extension ".js"

# **Node.js Get Started**

## Download Node.js

The official Node.js website has installation instructions for Node.js: [https://nodejs.org](https://nodejs.org/)

## Getting Started

Once you have downloaded and installed Node.js on your computer, let's try to display "Hello World" in a web browser.

Create a Node.js file named "myfirst.js", and add the following code:

myfirst.js

var http = require('http');  
  
http.createServer(function (req, res) {  
  res.writeHead(200, {'Content-Type': 'text/html'});  
  res.end('Hello World!');  
}).listen(8080);

Save the file on your computer: C:\Users\Your Name\myfirst.js

The code tells the computer to write "Hello World!" if anyone (e.g. a web browser) tries to access your computer on port 8080.

For now, you do not have to understand the code. It will be explained later.

## Command Line Interface

Node.js files must be initiated in the "Command Line Interface" program of your computer.

How to open the command line interface on your computer depends on the operating system. For Windows users, press the start button and look for "Command Prompt", or simply write "cmd" in the search field.

Navigate to the folder that contains the file "myfirst.js", the command line interface window should look something like this:

C:\Users\Your Name>\_

## Initiate the Node.js File

The file you have just created must be initiated by Node.js before any action can take place.

Start your command line interface, write node myfirst.js and hit enter:

Initiate "myfirst.js":

C:\Users\Your Name>node myfirst.js

Now, your computer works as a server!

If anyone tries to access your computer on port 8080, they will get a "Hello World!" message in return!

Start your internet browser, and type in the address: [http://localhost:8080](http://localhost:8080/)

# **Node.js Modules**

## What is a Module in Node.js?

Consider modules to be the same as JavaScript libraries.

A set of functions you want to include in your application.

## Built-in Modules

Node.js has a set of built-in modules which you can use without any further installation.

Look at our [Built-in Modules Reference](https://www.w3schools.com/nodejs/ref_modules.asp) for a complete list of modules.

## Include Modules

To include a module, use the require() function with the name of the module:

var http = require('http');

Now your application has access to the HTTP module, and is able to create a server:

http.createServer(function (req, res) {  
  res.writeHead(200, {'Content-Type': 'text/html'});  
  res.end('Hello World!');  
}).listen(8080);

## Create Your Own Modules

You can create your own modules, and easily include them in your applications.

The following example creates a module that returns a date and time object:

### Example[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

Create a module that returns the current date and time:

exports.myDateTime = function () {  
  return Date();  
};

Use the exports keyword to make properties and methods available outside the module file.

Save the code above in a file called "myfirstmodule.js"

Include Your Own Module

Now you can include and use the module in any of your Node.js files.

Example

Use the module "myfirstmodule" in a Node.js file:

var http = require('http');  
**var dt = require('./myfirstmodule');**  
http.createServer(function (req, res) {  
  res.writeHead(200, {'Content-Type': 'text/html'});  
  res.write("The date and time are currently: " + **dt.myDateTime()**);  
  res.end();  
}).listen(8080);

Notice that we use ./ to locate the module, that means that the module is located in the same folder as the Node.js file.

Save the code above in a file called "demo\_module.js", and initiate the file:

Initiate demo\_module.js:

C:\Users\Your Name>node demo\_module.js

If you have followed the same steps on your computer, you will see the same result as the example: [http://localhost:8080](http://localhost:8080/)

# **Node.js HTTP Module**

## The Built-in HTTP Module

Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP).

To include the HTTP module, use the require() method:

var http = require('http');

## Node.js as a Web Server

The HTTP module can create an HTTP server that listens to server ports and gives a response back to the client.

Use the createServer() method to create an HTTP server:

### Example[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

var http = require('http');  
  
//create a server object:  
http.createServer(function (req, res) {  
  res.write('Hello World!'); //write a response to the client  
  res.end(); //end the response  
}).listen(8080); //the server object listens on port 8080

The function passed into the http.createServer() method, will be executed when someone tries to access the computer on port 8080.

Save the code above in a file called "demo\_http.js", and initiate the file:

Initiate demo\_http.js:

C:\Users\Your Name>node demo\_http.js

If you have followed the same steps on your computer, you will see the same result as the example: [http://localhost:8080](http://localhost:8080/)

## Add an HTTP Header

If the response from the HTTP server is supposed to be displayed as HTML, you should include an HTTP header with the correct content type:

### Example

var http = require('http');  
http.createServer(function (req, res) {  
**res.writeHead(200, {'Content-Type': 'text/html'});**  res.write('Hello World!');  
  res.end();  
}).listen(8080);

The first argument of the res.writeHead() method is the status code, 200 means that all is OK, the second argument is an object containing the response headers.

## Read the Query String

The function passed into the http.createServer() has a req argument that represents the request from the client, as an object (http.IncomingMessage object).

This object has a property called "url" which holds the part of the url that comes after the domain name:

demo\_http\_url.js

var http = require('http');  
http.createServer(function (**req**, res) {  
  res.writeHead(200, {'Content-Type': 'text/html'});  
  res.write(**req.url**);  
  res.end();  
}).listen(8080);

Save the code above in a file called "demo\_http\_url.js" and initiate the file:

Initiate demo\_http\_url.js:

C:\Users\Your Name>node demo\_http\_url.js

If you have followed the same steps on your computer, you should see two different results when opening these two addresses:

<http://localhost:8080/summer>

Will produce this result:

/summer

<http://localhost:8080/winter>

Will produce this result:

/winter

## Split the Query String

There are built-in modules to easily split the query string into readable parts, such as the URL module.

### Example

Split the query string into readable parts:

var http = require('http');  
**var url = require('url');**  
http.createServer(function (req, res) {  
  res.writeHead(200, {'Content-Type': 'text/html'});  
**var q = url.parse(req.url, true).query;**  var txt = **q.year** + " " + **q.month**;  
  res.end(txt);  
}).listen(8080);

Save the code above in a file called "demo\_querystring.js" and initiate the file:

Initiate demo\_querystring.js:

C:\Users\Your Name>node demo\_querystring.js

The address:

<http://localhost:8080/?year=2017&month=July>

Will produce this result:

2017 July

Read more about the URL module in the [Node.js URL Module](https://www.w3schools.com/nodejs/nodejs_url.asp) chapter.

# **Node.js File System Module**

## Node.js as a File Server

The Node.js file system module allows you to work with the file system on your computer.

To include the File System module, use the require() method:

var fs = require('fs');

Common use for the File System module:

* Read files
* Create files
* Update files
* Delete files
* Rename files

## Read Files

The fs.readFile() method is used to read files on your computer.

Assume we have the following HTML file (located in the same folder as Node.js):

demofile1.html

<html>  
<body>  
<h1>My Header</h1>  
<p>My paragraph.</p>  
</body>  
</html>

Create a Node.js file that reads the HTML file, and return the content:

### Example[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

var http = require('http');  
var fs = require('fs');  
http.createServer(function (req, res) {  
**fs.readFile('demofile1.html', function(err, data) {**    res.writeHead(200, {'Content-Type': 'text/html'});  
    res.write(data);  
    return res.end();  
  });  
}).listen(8080);

Save the code above in a file called "demo\_readfile.js", and initiate the file:

Initiate demo\_readfile.js:

C:\Users\Your Name>node demo\_readfile.js

If you have followed the same steps on your computer, you will see the same result as the example: [http://localhost:8080](http://localhost:8080/)

## Create Files

The File System module has methods for creating new files:

* fs.appendFile()
* fs.open()
* fs.writeFile()

The fs.appendFile() method appends specified content to a file. If the file does not exist, the file will be created:

### Example

Create a new file using the appendFile() method:

var fs = require('fs');  
  
fs.appendFile('mynewfile1.txt', 'Hello content!', function (err) {  
  if (err) throw err;  
  console.log('Saved!');  
});

The fs.open() method takes a "flag" as the second argument, if the flag is "w" for "writing", the specified file is opened for writing. If the file does not exist, an empty file is created:

### Example

Create a new, empty file using the open() method:

var fs = require('fs');  
  
fs.open('mynewfile2.txt', 'w', function (err, file) {  
  if (err) throw err;  
  console.log('Saved!');  
});

The fs.writeFile() method replaces the specified file and content if it exists. If the file does not exist, a new file, containing the specified content, will be created:

### Example

Create a new file using the writeFile() method:

var fs = require('fs');  
  
fs.writeFile('mynewfile3.txt', 'Hello content!', function (err) {  
  if (err) throw err;  
  console.log('Saved!');  
});

## Update Files

The File System module has methods for updating files:

* fs.appendFile()
* fs.writeFile()

The fs.appendFile() method appends the specified content at the end of the specified file:

### Example

Append "This is my text." to the end of the file "mynewfile1.txt":

var fs = require('fs');  
  
fs.appendFile('mynewfile1.txt', ' This is my text.', function (err) {  
  if (err) throw err;  
  console.log('Updated!');  
});

The fs.writeFile() method replaces the specified file and content:

### Example

Replace the content of the file "mynewfile3.txt":

var fs = require('fs');  
  
fs.writeFile('mynewfile3.txt', 'This is my text', function (err) {  
  if (err) throw err;  
  console.log('Replaced!');  
});

## Delete Files

To delete a file with the File System module,  use the fs.unlink() method.

The fs.unlink() method deletes the specified file:

### Example

Delete "mynewfile2.txt":

var fs = require('fs');  
  
fs.unlink('mynewfile2.txt', function (err) {  
  if (err) throw err;  
  console.log('File deleted!');  
});

## Rename Files

To rename a file with the File System module,  use the fs.rename() method.

The fs.rename() method renames the specified file:

### Example

Rename "mynewfile1.txt" to "myrenamedfile.txt":

var fs = require('fs');  
  
fs.rename('mynewfile1.txt', 'myrenamedfile.txt', function (err) {  
  if (err) throw err;  
  console.log('File Renamed!');  
});

## Upload Files

You can also use Node.js to upload files to your computer.

Read how in our [Node.js Upload Files](https://www.w3schools.com/nodejs/nodejs_uploadfiles.asp) chapter.

# **Node.js URL Module**

## The Built-in URL Module

The URL module splits up a web address into readable parts.

To include the URL module, use the require() method:

var url = require('url');

Parse an address with the url.parse() method, and it will return a URL object with each part of the address as properties:

### Example

Split a web address into readable parts:

var url = require('url');  
var adr = 'http://localhost:8080/default.htm?year=2017&month=february';  
var q = url.parse(adr, true);  
  
console.log(q.host); //returns 'localhost:8080'  
console.log(q.pathname); //returns '/default.htm'  
console.log(q.search); //returns '?year=2017&month=february'  
  
var qdata = q.query; //returns an object: { year: 2017, month: 'february' }  
console.log(qdata.month); //returns 'february'

## Node.js File Server

Now we know how to parse the query string, and in the previous chapter we learned how to make Node.js behave as a file server. Let us combine the two, and serve the file requested by the client.

Create two html files and save them in the same folder as your node.js files.

summer.html

<!DOCTYPE html>  
<html>  
<body>  
<h1>Summer</h1>  
<p>I love the sun!</p>  
</body>  
</html>

winter.html

<!DOCTYPE html>  
<html>  
<body>  
<h1>Winter</h1>  
<p>I love the snow!</p>  
</body>  
</html>

Create a Node.js file that opens the requested file and returns the content to the client. If anything goes wrong, throw a 404 error:

demo\_fileserver.js:

var http = require('http');  
var url = require('url');  
var fs = require('fs');  
  
http.createServer(function (req, res) {  
  var q = url.parse(req.url, true);  
  var filename = "." + q.pathname;  
  fs.readFile(filename, function(err, data) {  
    if (err) {  
      res.writeHead(404, {'Content-Type': 'text/html'});  
      return res.end("404 Not Found");  
    }   
    res.writeHead(200, {'Content-Type': 'text/html'});  
    res.write(data);  
    return res.end();  
  });  
}).listen(8080);

Remember to initiate the file:

Initiate demo\_fileserver.js:

C:\Users\Your Name>node demo\_fileserver.js

If you have followed the same steps on your computer, you should see two different results when opening these two addresses:

<http://localhost:8080/summer.html>

Will produce this result:

# **Summer**

I love the sun!

<http://localhost:8080/winter.html>

Will produce this result:

# **Winter**

I love the snow!

# **Node.js NPM**

## What is NPM?

NPM is a package manager for Node.js packages, or modules if you like.

[www.npmjs.com](https://www.npmjs.com/) hosts thousands of free packages to download and use.

The NPM program is installed on your computer when you install Node.js

NPM is already ready to run on your computer!

## What is a Package?

A package in Node.js contains all the files you need for a module.

Modules are JavaScript libraries you can include in your project.

## Download a Package

Downloading a package is very easy.

Open the command line interface and tell NPM to download the package you want.

I want to download a package called "upper-case":

Download "upper-case":

C:\Users\Your Name>npm install upper-case

Now you have downloaded and installed your first package!

NPM creates a folder named "node\_modules", where the package will be placed. All packages you install in the future will be placed in this folder.

My project now has a folder structure like this:

C:\Users\My Name\node\_modules\upper-case

## Using a Package

Once the package is installed, it is ready to use.

Include the "upper-case" package the same way you include any other module:

var uc = require('upper-case');

Create a Node.js file that will convert the output "Hello World!" into upper-case letters:

### Example[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

var http = require('http');  
var uc = require('upper-case');  
http.createServer(function (req, res) {  
  res.writeHead(200, {'Content-Type': 'text/html'});  
  res.write(**uc.upperCase("Hello World!")**);  
  res.end();  
}).listen(8080);

Save the code above in a file called "demo\_uppercase.js", and initiate the file:

Initiate demo\_uppercase:

C:\Users\Your Name>node demo\_uppercase.js

If you have followed the same steps on your computer, you will see the same result as the example: [http://localhost:8080](http://localhost:8080/)

# **Node.js Events**

Node.js is perfect for event-driven applications.

## Events in Node.js

Every action on a computer is an event. Like when a connection is made or a file is opened.

Objects in Node.js can fire events, like the readStream object fires events when opening and closing a file:

### Example[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

var fs = require('fs');  
var rs = fs.createReadStream('./demofile.txt');  
rs.on('open', function () {  
  console.log('The file is open');  
});

## Events Module

Node.js has a built-in module, called "Events", where you can create-, fire-, and listen for- your own events.

To include the built-in Events module use the require() method. In addition, all event properties and methods are an instance of an EventEmitter object. To be able to access these properties and methods, create an EventEmitter object:

var events = require('events');  
var eventEmitter = new events.EventEmitter();

## The EventEmitter Object

You can assign event handlers to your own events with the EventEmitter object.

In the example below we have created a function that will be executed when a "scream" event is fired.

To fire an event, use the emit() method.

### Example

var events = require('events');  
var eventEmitter = new events.EventEmitter();  
  
//Create an event handler:  
var myEventHandler = function () {  
  console.log('I hear a scream!');  
}  
  
//Assign the event handler to an event:  
eventEmitter.on('scream', myEventHandler);  
  
//Fire the 'scream' event:  
eventEmitter.emit('scream');

# **Node.js Upload Files**

## The Formidable Module

There is a very good module for working with file uploads, called "Formidable".

The Formidable module can be downloaded and installed using NPM:

C:\Users\Your Name>npm install formidable

After you have downloaded the Formidable module, you can include the module in any application:

var formidable = require('formidable');

## Upload Files

Now you are ready to make a web page in Node.js that lets the user upload files to your computer:

## Step 1: Create an Upload Form

Create a Node.js file that writes an HTML form, with an upload field:

### Example[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

This code will produce an HTML form:

var http = require('http');  
  
http.createServer(function (req, res) {  
  res.writeHead(200, {'Content-Type': 'text/html'});  
  res.write('<form action="fileupload" method="post" enctype="multipart/form-data">');  
  res.write('<input type="file" name="filetoupload"><br>');  
  res.write('<input type="submit">');  
  res.write('</form>');  
  return res.end();  
}).listen(8080);

## Step 2: Parse the Uploaded File

Include the Formidable module to be able to parse the uploaded file once it reaches the server.

When the file is uploaded and parsed, it gets placed on a temporary folder on your computer.

### Example

The file will be uploaded, and placed on a temporary folder:

var http = require('http');  
**var formidable = require('formidable');**  
http.createServer(function (req, res) {  
**if (req.url == '/fileupload') {  
    var form = new formidable.IncomingForm();  
    form.parse(req, function (err, fields, files) {  
      res.write('File uploaded');  
      res.end();  
    });**  } else {  
    res.writeHead(200, {'Content-Type': 'text/html'});  
    res.write('<form action="fileupload" method="post" enctype="multipart/form-data">');  
    res.write('<input type="file" name="filetoupload"><br>');  
    res.write('<input type="submit">');  
    res.write('</form>');  
    return res.end();  
  }  
}).listen(8080);

## Step 3: Save the File

When a file is successfully uploaded to the server, it is placed on a temporary folder.

The path to this directory can be found in the "files" object, passed as the third argument in the parse() method's callback function.

To move the file to the folder of your choice, use the File System module, and rename the file:

### Example

Include the fs module, and move the file to the current folder:

var http = require('http');  
var formidable = require('formidable');  
**var fs = require('fs');**  
http.createServer(function (req, res) {  
  if (req.url == '/fileupload') {  
    var form = new formidable.IncomingForm();  
    form.parse(req, function (err, fields, files) {  
**var oldpath = files.filetoupload.filepath;  
      var newpath = 'C:/Users/Your Name/' + files.filetoupload.originalFilename;  
      fs.rename(oldpath, newpath, function (err) {  
        if (err) throw err;  
        res.write('File uploaded and moved!');  
        res.end();  
      });** });  
  } else {  
    res.writeHead(200, {'Content-Type': 'text/html'});  
    res.write('<form action="fileupload" method="post" enctype="multipart/form-data">');  
    res.write('<input type="file" name="filetoupload"><br>');  
    res.write('<input type="submit">');  
    res.write('</form>');  
    return res.end();  
  }  
}).listen(8080);

# **Node.js Send an Email**

## The Nodemailer Module

The Nodemailer module makes it easy to send emails from your computer.

The Nodemailer module can be downloaded and installed using npm:

C:\Users\Your Name>npm install nodemailer

After you have downloaded the Nodemailer module, you can include the module in any application:

var nodemailer = require('nodemailer');

## Send an Email

Now you are ready to send emails from your server.

Use the username and password from your selected email provider to send an email. This tutorial will show you how to use your Gmail account to send an email:

### Example[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

var nodemailer = require('nodemailer');  
  
var transporter = nodemailer.createTransport({  
  service: 'gmail',  
  auth: {  
    user: 'youremail@gmail.com',  
    pass: 'yourpassword'  
  }  
});  
  
var mailOptions = {  
  from: 'youremail@gmail.com',  
  to: 'myfriend@yahoo.com',  
  subject: 'Sending Email using Node.js',  
  text: 'That was easy!'  
};  
  
transporter.sendMail(mailOptions, function(error, info){  
  if (error) {  
    console.log(error);  
  } else {  
    console.log('Email sent: ' + info.response);  
  }  
});

And that's it! Now your server is able to send emails.

## Multiple Receivers

To send an email to more than one receiver, add them to the "to" property of the mailOptions object, separated by commas:

### Example

Send email to more than one address:

var mailOptions = {  
  from: 'youremail@gmail.com',  
**to:**'***myfriend@yahoo.com*, *myotherfriend@yahoo.com*',**  subject: 'Sending Email using Node.js',  
  text: 'That was easy!'  
}

## Send HTML

To send HTML formatted text in your email, use the "html" property instead of the "text" property:

### Example

Send email containing HTML:

var mailOptions = {  
  from: 'youremail@gmail.com',  
  to: 'myfriend@yahoo.com',  
  subject: 'Sending Email using Node.js',  
**html: '<h1>Welcome</h1><p>That was easy!</p>'**}

# **Node.js MySQL**

Node.js can be used in database applications.

One of the most popular databases is MySQL.

## MySQL Database

To be able to experiment with the code examples, you should have MySQL installed on your computer.

You can download a free MySQL database at <https://www.mysql.com/downloads/>.

## Install MySQL Driver

Once you have MySQL up and running on your computer, you can access it by using Node.js.

To access a MySQL database with Node.js, you need a MySQL driver. This tutorial will use the "mysql" module, downloaded from NPM.

To download and install the "mysql" module, open the Command Terminal and execute the following:

C:\Users\Your Name>npm install mysql

Now you have downloaded and installed a mysql database driver.

Node.js can use this module to manipulate the MySQL database:

var mysql = require('mysql');

## Create Connection

Start by creating a connection to the database.

Use the username and password from your MySQL database.

demo\_db\_connection.js

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");  
});

Save the code above in a file called "demo\_db\_connection.js" and run the file:

Run "demo\_db\_connection.js"

C:\Users\Your Name>node demo\_db\_connection.js

Which will give you this result:

Connected!

Now you can start querying the database using SQL statements.

## Query a Database

Use SQL statements to read from (or write to) a MySQL database. This is also called "to query" the database.

The connection object created in the example above, has a method for querying the database:

con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");  
**con.query(*sql*, function (err, result) {  
    if (err) throw err;  
    console.log("Result: " + result);  
  });**});

The query method takes an sql statements as a parameter and returns the result.

Learn how to read, write, delete, and update a database in the next chapters.

Read more about SQL statements in our [SQL Tutorial](https://www.w3schools.com/sql/default.asp).

# **Node.js MySQL Create Database**

## Creating a Database

To create a database in MySQL, use the "CREATE DATABASE" statement:

## Example

Create a database named "mydb":

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");  
  con.query("**CREATE DATABASE mydb**", function (err, result) {  
    if (err) throw err;  
    console.log("Database created");  
  });  
});

Save the code above in a file called "demo\_create\_db.js" and run the file:

Run "demo\_create\_db.js"

C:\Users\Your Name>node demo\_create\_db.js

Which will give you this result:

Connected!  
Database created

# **Node.js MySQL Create Table**

## Creating a Table

To create a table in MySQL, use the "CREATE TABLE" statement.

Make sure you define the name of the database when you create the connection:

### Example[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

Create a table named "customers":

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
**database: "mydb"**  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");  
  var sql = "**CREATE TABLE customers (name VARCHAR(255), address VARCHAR(255)**)";  
  con.query(sql, function (err, result) {  
    if (err) throw err;  
    console.log("Table created");  
  });  
});

Save the code above in a file called "demo\_create\_table.js" and run the file:

Run "demo\_create\_table.js"

C:\Users\Your Name>node demo\_create\_table.js

Which will give you this result:

Connected!  
Table created

ADVERTISEMENT

## Primary Key

When creating a table, you should also create a column with a unique key for each record.

This can be done by defining a column as "INT AUTO\_INCREMENT PRIMARY KEY" which will insert a unique number for each record. Starting at 1, and increased by one for each record.

### Example

Create primary key when creating the table:

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");  
  var sql = "CREATE TABLE customers (**id INT AUTO\_INCREMENT PRIMARY KEY**, name VARCHAR(255), address VARCHAR(255))";  
  con.query(sql, function (err, result) {  
    if (err) throw err;  
    console.log("Table created");  
  });  
});

If the table already exists, use the ALTER TABLE keyword:

### Example

Create primary key on an existing table:

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");  
  var sql = "**ALTER TABLE customers ADD COLUMN id INT AUTO\_INCREMENT PRIMARY KEY**";  
  con.query(sql, function (err, result) {  
    if (err) throw err;  
    console.log("Table altered");  
  });  
});

# **Node.js MySQL Insert Into**

## Insert Into Table

To fill a table in MySQL, use the "INSERT INTO" statement.

### Example[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

Insert a record in the "customers" table:

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");  
**var sql = "INSERT INTO customers (name, address) VALUES ('Company Inc', 'Highway 37')";**  con.query(sql, function (err, result) {  
    if (err) throw err;  
    console.log("1 record inserted");  
  });  
});

Save the code above in a file called "demo\_db\_insert.js", and run the file:

Run "demo\_db\_insert.js"

C:\Users\Your Name>node demo\_db\_insert.js

Which will give you this result:

Connected!  
1 record inserted

ADVERTISEMENT

## Insert Multiple Records

To insert more than one record, make an array containing the values, and insert a question mark in the sql, which will be replaced by the value array:  
INSERT INTO customers (name, address) VALUES ?

### Example

Fill the "customers" table with data:

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");  
  var sql = "INSERT INTO customers (name, address) VALUES ?";  
  var values = [  
    ['John', 'Highway 71'],  
    ['Peter', 'Lowstreet 4'],  
    ['Amy', 'Apple st 652'],  
    ['Hannah', 'Mountain 21'],  
    ['Michael', 'Valley 345'],  
    ['Sandy', 'Ocean blvd 2'],  
    ['Betty', 'Green Grass 1'],  
    ['Richard', 'Sky st 331'],  
    ['Susan', 'One way 98'],  
    ['Vicky', 'Yellow Garden 2'],  
    ['Ben', 'Park Lane 38'],  
    ['William', 'Central st 954'],  
    ['Chuck', 'Main Road 989'],  
    ['Viola', 'Sideway 1633']  
  ];  
  con.query(sql,**[values]**, function (err, result) {  
    if (err) throw err;  
    console.log("Number of records inserted: " + result.affectedRows);  
  });  
});

Save the code above in a file called "demo\_db\_insert\_multple.js", and run the file:

Run "demo\_db\_insert\_multiple.js"

C:\Users\Your Name>node demo\_db\_insert\_multiple.js

Which will give you this result:

Connected!  
Number of records inserted: 14

## The Result Object

When executing a query, a result object is returned.

The result object contains information about how the query affected the table.

The result object returned from the example above looks like this:

{  
  fieldCount: 0,  
  affectedRows: 14,  
  insertId: 0,  
  serverStatus: 2,  
  warningCount: 0,  
  message: '\'Records:14  Duplicated: 0  Warnings: 0',  
  protocol41: true,  
  changedRows: 0  
}

The values of the properties can be displayed like this:

### Example

Return the number of affected rows:

console.log(result.affectedRows)

Which will produce this result:

14

## Get Inserted ID

For tables with an auto increment id field, you can get the id of the row you just inserted by asking the result object.

**Note:** To be able to get the inserted id, **only one row** can be inserted.

### Example

Insert a record in the "customers" table, and return the ID:

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  var sql = "INSERT INTO customers (name, address) VALUES ('Michelle', 'Blue Village 1')";  
  con.query(sql, function (err, result) {  
    if (err) throw err;  
    console.log("1 record inserted, ID: " + **result.insertId**);  
  });  
});

Save the code above in a file called "demo\_db\_insert\_id.js", and run the file:

Run "demo\_db\_insert\_id.js"

C:\Users\Your Name>node demo\_db\_insert\_id.js

Which will give you something like this in return:

1 record inserted, ID: 15

# **Node.js MySQL Select From**

## Selecting From a Table

To select data from a table in MySQL, use the "SELECT" statement.

### Example[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

Select all records from the "customers" table, and display the result object:

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  con.query("**SELECT \* FROM customers**", function (err, result, fields) {  
    if (err) throw err;  
    console.log(result);  
  });  
});

**SELECT \*** will return all columns

Save the code above in a file called "demo\_db\_select.js" and run the file:

Run "demo\_db\_select.js"

C:\Users\Your Name>node demo\_db\_select.js

Which will give you this result:

[  
  { id: 1, name: 'John', address: 'Highway 71'},  
  { id: 2, name: 'Peter', address: 'Lowstreet 4'},  
  { id: 3, name: 'Amy', address: 'Apple st 652'},  
  { id: 4, name: 'Hannah', address: 'Mountain 21'},  
  { id: 5, name: 'Michael', address: 'Valley 345'},  
  { id: 6, name: 'Sandy', address: 'Ocean blvd 2'},  
  { id: 7, name: 'Betty', address: 'Green Grass 1'},  
  { id: 8, name: 'Richard', address: 'Sky st 331'},  
  { id: 9, name: 'Susan', address: 'One way 98'},  
  { id: 10, name: 'Vicky', address: 'Yellow Garden 2'},  
  { id: 11, name: 'Ben', address: 'Park Lane 38'},  
  { id: 12, name: 'William', address: 'Central st 954'},  
  { id: 13, name: 'Chuck', address: 'Main Road 989'},  
  { id: 14, name: 'Viola', address: 'Sideway 1633'}  
]

## Selecting Columns

To select only some of the columns in a table, use the "SELECT" statement followed by the column name.

### Example

Select name and address from the "customers" table, and display the return object:

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  con.query("**SELECT name, address FROM customers**", function (err, result, fields) {  
    if (err) throw err;  
    console.log(result);  
  });  
});

Save the code above in a file called "demo\_db\_select2.js" and run the file:

Run "demo\_db\_select2.js"

C:\Users\Your Name>node demo\_db\_select2.js

Which will give you this result:

[  
  { name: 'John', address: 'Highway 71'},  
  { name: 'Peter', address: 'Lowstreet 4'},  
  { name: 'Amy', address: 'Apple st 652'},  
  { name: 'Hannah', address: 'Mountain 21'},  
  { name: 'Michael', address: 'Valley 345'},  
  { name: 'Sandy', address: 'Ocean blvd 2'},  
  { name: 'Betty', address: 'Green Grass 1'},  
  { name: 'Richard', address: 'Sky st 331'},  
  { name: 'Susan', address: 'One way 98'},  
  { name: 'Vicky', address: 'Yellow Garden 2'},  
  { name: 'Ben', address: 'Park Lane 38'},  
  { name: 'William', address: 'Central st 954'},  
  { name: 'Chuck', address: 'Main Road 989'},  
  { name: 'Viola', address: 'Sideway 1633'}  
]

## The Result Object

As you can see from the result of the example above, the result object is an array containing each row as an object.

To return e.g. the address of the third record, just refer to the third array object's address property:

### Example

Return the address of the third record:

console.log(result[2].address);

Which will produce this result:

Apple st 652

## The Fields Object

The third parameter of the callback function is an array containing information about each field in the result.

### Example

Select all records from the "customers" table, and display the fields object:

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  con.query("SELECT name, address FROM customers", function (err, result, **fields**) {  
    if (err) throw err;  
    console.log(**fields**);  
  });  
});

Save the code above in a file called "demo\_db\_select\_fields.js" and run the file:

Run "demo\_db\_select\_fields.js"

C:\Users\Your Name>node demo\_db\_select\_fields.js

Which will give you this result:

[  
  {  
    catalog: 'def',  
    db: 'mydb',  
    table: 'customers',  
    orgTable: 'customers',  
    name: 'name',  
    orgName: 'name',  
    charsetNr: 33,  
    length: 765,  
    type: 253,  
    flags: 0,  
    decimals: 0,  
    default: undefined,  
    zeroFill: false,  
    protocol41: true  
  },  
  {  
    catalog: 'def',  
    db: 'mydb',  
    table: 'customers',  
    orgTable: 'customers',  
    name: 'address',  
    orgName: 'address',  
    charsetNr: 33,  
    length: 765,  
    type: 253,  
    flags: 0,  
    decimals: 0,  
    default: undefined,  
    zeroFill: false,  
    protocol41: true  
  }  
]

As you can see from the result of the example above, the fields object is an array containing information about each field as an object.

To return e.g. the name of the second field, just refer to the second array item's name property:

### Example

Return the name of the second field:

console.log(fields[1].name);

Which will produce this result:

address

# **Node.js MySQL Where**

## Select With a Filter

When selecting records from a table, you can filter the selection by using the "WHERE" statement:

### Example[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

Select record(s) with the address "Park Lane 38":

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  con.query("SELECT \* FROM customers **WHERE address = 'Park Lane 38'**", function (err, result) {  
    if (err) throw err;  
    console.log(result);  
  });  
});

Save the code above in a file called "demo\_db\_where.js" and run the file:

Run "demo\_db\_where.js"

C:\Users\Your Name>node demo\_db\_where.js

Which will give you this result:

[  
  { id: 11, name: 'Ben', address: 'Park Lane 38'}  
]

## Wildcard Characters

You can also select the records that starts, includes, or ends with a given letter or phrase.

Use the '%' wildcard to represent zero, one or multiple characters:

### Example

Select records where the address starts with the letter 'S':

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  con.query("SELECT \* FROM customers WHERE address **LIKE 'S%'**", function (err, result) {  
    if (err) throw err;  
    console.log(result);  
  });  
});

Save the code above in a file called "demo\_db\_where\_s.js" and run the file:

Run "demo\_db\_where\_s.js"

C:\Users\Your Name>node demo\_db\_where\_s.js

Which will give you this result:

[  
  { id: 8, name: 'Richard', address: 'Sky st 331'},  
  { id: 14, name: 'Viola', address: 'Sideway 1633'}  
]

## Escaping Query Values

When query values are variables provided by the user, you should escape the values.

This is to prevent SQL injections, which is a common web hacking technique to destroy or misuse your database.

The MySQL module has methods to escape query values:

### Example

Escape query values by using the mysql.escape() method:

var adr = 'Mountain 21';  
var sql = 'SELECT \* FROM customers WHERE address = ' + mysql.escape(adr);  
con.query(sql, function (err, result) {  
  if (err) throw err;  
  console.log(result);  
});

You can also use a ? as a placeholder for the values you want to escape.

In this case, the variable is sent as the second parameter in the query() method:

### Example

Escape query values by using the placeholder ? method:

var adr = 'Mountain 21';  
var sql = 'SELECT \* FROM customers WHERE address = ?';  
con.query(sql,**[adr]**, function (err, result) {  
  if (err) throw err;  
  console.log(result);  
});

If you have multiple placeholders, the array contains multiple values, in that order:

### Example

Multiple placeholders:

var name = 'Amy';  
var adr = 'Mountain 21';  
var sql = 'SELECT \* FROM customers WHERE name = ? OR address = ?';  
con.query(sql,**[name, adr]**, function (err, result) {  
  if (err) throw err;  
  console.log(result);  
});

## ORDER BY DESC

Use the DESC keyword to sort the result in a descending order.

### Example

Sort the result reverse alphabetically by name:

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  con.query("SELECT \* FROM customers **ORDER BY name DESC**", function (err, result) {  
    if (err) throw err;  
    console.log(result);  
  });  
});

Save the code above in a file called "demo\_db\_orderby\_desc.js" and run the file:

Run "demo\_db\_orderby\_desc.js"

C:\Users\Your Name>node demo\_db\_orderby\_desc.js

Which will give you this result:

[  
  { id: 12, name: 'William', address: 'Central st 954'},  
  { id: 14, name: 'Viola', address: 'Sideway 1633'},  
  { id: 10, name: 'Vicky', address: 'Yellow Garden 2'},  
  { id: 9, name: 'Susan', address: 'One way 98'},  
  { id: 6, name: 'Sandy', address: 'Ocean blvd 2'},  
  { id: 8, name: 'Richard', address: 'Sky st 331'},  
  { id: 2, name: 'Peter', address: 'Lowstreet 4'},  
  { id: 5, name: 'Michael', address: 'Valley 345'},  
  { id: 1, name: 'John', address: 'Higheay 71'},  
  { id: 4, name: 'Hannah', address: 'Mountain 21'},  
  { id: 13, name: 'Chuck', address: 'Main Road 989'},  
  { id: 7, name: 'Betty', address: 'Green Grass 1'},  
  { id: 11, name: 'Ben', address: 'Park Lane 38'},  
  { id: 3, name: 'Amy', address: 'Apple st 652'}  
]

# **Node.js MySQL Delete**

## Delete Record

You can delete records from an existing table by using the "DELETE FROM" statement:

### Example[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

Delete any record with the address "Mountain 21":

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
**var sql = "DELETE FROM customers WHERE address = 'Mountain 21'";**  con.query(**sql,** function (err, result) {  
    if (err) throw err;  
    console.log("Number of records deleted: " + result.affectedRows);  
  });  
});

**Notice the WHERE clause in the DELETE syntax:** The WHERE clause specifies which record or records that should be deleted. If you omit the WHERE clause, all records will be deleted!

Save the code above in a file called "demo\_db\_delete.js" and run the file:

Run "demo\_db\_delete.js"

C:\Users\Your Name>node demo\_db\_delete.js

Which will give you this result:

Number of records deleted: 1

## The Result Object

When executing a query, a result object is returned.

The result object contains information about how the query affected the table.

The result object returned from the example above looks like this:

{  
  fieldCount: 0,  
  affectedRows: 1,  
  insertId: 0,  
  serverStatus: 34,  
  warningCount: 0,  
  message: '',  
  protocol41: true,  
  changedRows: 0  
}

The values of the properties can be displayed like this:

### Example

Return the number of affected rows:

console.log(result.affectedRows)

Which will produce this result:

1

# **Node.js MySQL Drop Table**

## Delete a Table

You can delete an existing table by using the "DROP TABLE" statement:

### Example[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

Delete the table "customers":

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
**var sql = "DROP TABLE customers";**  con.query(**sql,** function (err, result) {  
    if (err) throw err;  
    console.log("Table deleted");  
  });  
});

Save the code above in a file called "demo\_db\_drop\_table.js" and run the file:

Run "demo\_db\_drop\_table.js"

C:\Users\Your Name>node demo\_db\_drop\_table.js

Which will give you this result:

Table deleted

## Drop Only if Exist

If the the table you want to delete is already deleted, or for any other reason does not exist, you can use the IF EXISTS keyword to avoid getting an error.

### Example

Delete the table "customers" if it exists:

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
**var sql = "DROP TABLE IF EXISTS customers";**  con.query(**sql,** function (err, result) {  
    if (err) throw err;  
    console.log(result);  
  });  
});

Save the code above in a file called "demo\_db\_drop\_table\_if.js" and run the file:

Run "demo\_db\_drop\_table\_if.js"

C:\Users\Your Name>node demo\_db\_drop\_table\_if.js

If the table exist, the result object will look like this:

{  
  fieldCount: 0,  
  affectedRows: 0,  
  insertId: 0,  
  serverstatus: 2,  
  warningCount: 0,  
  message: '',  
  protocol41: true,  
  changedRows: 0  
}

If the table does not exist, the result object will look like this:

{  
  fieldCount: 0,  
  affectedRows: 0,  
  insertId: 0,  
  serverstatus: 2,  
  warningCount: 1,  
  message: '',  
  protocol41: true,  
  changedRows: 0  
}

As you can see the only differnce is that the warningCount property is set to 1 if the table does not exist.

# **Node.js MySQL Update**

## Update Table

You can update existing records in a table by using the "UPDATE" statement:

### Example[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

Overwrite the address column from "Valley 345" to "Canyon 123":

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
**var sql = "UPDATE customers SET address = 'Canyon 123' WHERE address = 'Valley 345'";**  con.query(**sql,** function (err, result) {  
    if (err) throw err;  
    console.log(result.affectedRows + " record(s) updated");  
  });  
});

**Notice the WHERE clause in the UPDATE syntax:** The WHERE clause specifies which record or records that should be updated. If you omit the WHERE clause, all records will be updated!

Save the code above in a file called "demo\_db\_update.js" and run the file:

Run "demo\_db\_update.js"

C:\Users\Your Name>node demo\_db\_update.js

Which will give you this result:

1 record(s) updated

## The Result Object

When executing a query, a result object is returned.

The result object contains information about how the query affected the table.

The result object returned from the example above looks like this:

{  
  fieldCount: 0,  
  affectedRows: 1,  
  insertId: 0,  
  serverStatus: 34,  
  warningCount: 0,  
  message: '(Rows matched: 1 Changed: 1 Warnings: 0',  
  protocol41: true,  
  changedRows: 1  
}

The values of the properties can be displayed like this:

### Example

Return the number of affected rows:

console.log(result.affectedRows)

Which will produce this result:

1

# **Node.js MySQL Limit**

## Limit the Result

You can limit the number of records returned from the query, by using the "LIMIT" statement:

### Example[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

Select the 5 first records in the "customers" table:

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  var sql = "SELECT \* FROM customers **LIMIT 5**";  
  con.query(sql, function (err, result) {  
    if (err) throw err;  
    console.log(result);  
  });  
});

Save the code above in a file called "demo\_db\_limit.js" and run the file:

Run "demo\_db\_limit.js"

C:\Users\Your Name>node demo\_db\_limit.js

Which will give you this result:

[  
  { id: 1, name: 'John', address: 'Highway 71'},  
  { id: 2, name: 'Peter', address: 'Lowstreet 4'},  
  { id: 3, name: 'Amy', address: 'Apple st 652'},  
  { id: 4, name: 'Hannah', address: 'Mountain 21'},  
  { id: 5, name: 'Michael', address: 'Valley 345'}  
]

## Start From Another Position

If you want to return five records, starting from the third record, you can use the "OFFSET" keyword:

### Example

Start from position 3, and return the next 5 records:

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  var sql = "SELECT \* FROM customers **LIMIT 5 OFFSET 2**";  
  con.query(sql, function (err, result) {  
    if (err) throw err;  
    console.log(result);  
  });  
});

**Note:** "OFFSET 2", means starting from the third position, not the second!

Save the code above in a file called "demo\_db\_offset.js" and run the file:

Run "demo\_db\_offset.js"

C:\Users\Your Name>node demo\_db\_offset.js

Which will give you this result:

[  
  { id: 3, name: 'Amy', address: 'Apple st 652'},  
  { id: 4, name: 'Hannah', address: 'Mountain 21'},  
  { id: 5, name: 'Michael', address: 'Valley 345'},  
  { id: 6, name: 'Sandy', address: 'Ocean blvd 2'},  
  { id: 7, name: 'Betty', address: 'Green Grass 1'}  
]

## Shorter Syntax

You can also write your SQL statement like this "LIMIT 2, 5" which returns the same as the offset example above:

### Example

Start from position 3, and return the next 5 records:

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  var sql = "SELECT \* FROM customers **LIMIT 2, 5**";  
  con.query(sql, function (err, result) {  
    if (err) throw err;  
    console.log(result);  
  });  
});

**Note:** The numbers are reversed: "LIMIT 2, 5" is the same as "LIMIT 5 OFFSET 2"

# **Node.js MySQL Join**

## Join Two or More Tables

You can combine rows from two or more tables, based on a related column between them, by using a JOIN statement.

Consider you have a "users" table and a "products" table:

### users[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

[  
  { id: 1, name: 'John', favorite\_product: 154},  
  { id: 2, name: 'Peter', favorite\_product: 154},  
  { id: 3, name: 'Amy', favorite\_product: 155},  
  { id: 4, name: 'Hannah', favorite\_product:},  
  { id: 5, name: 'Michael', favorite\_product:}  
]

### products

[  
  { id: 154, name: 'Chocolate Heaven' },  
  { id: 155, name: 'Tasty Lemons' },  
  { id: 156, name: 'Vanilla Dreams' }  
]

These two tables can be combined by using users' favorite\_product field and products' id field.

### Example

Select records with a match in both tables:

var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  var sql = "SELECT users.name AS user, products.name AS favorite FROM users **JOIN products ON users.favorite\_product = products.id**";  
  con.query(sql, function (err, result) {  
    if (err) throw err;  
    console.log(result);  
  });  
});

**Note:** You can use INNER JOIN instead of JOIN. They will both give you the same result.

Save the code above in a file called "demo\_db\_join.js" and run the file:

Run "demo\_db\_join.js"

C:\Users\Your Name>node demo\_db\_join.js

Which will give you this result:

[  
  { user: 'John', favorite: 'Chocolate Heaven' },  
  { user: 'Peter', favorite: 'Chocolate Heaven' },  
  { user: 'Amy', favorite: 'Tasty Lemons' }  
]

As you can see from the result above, only the records with a match in both tables are returned.

## Left Join

If you want to return all users, no matter if they have a favorite product or not, use the LEFT JOIN statement:

### Example

Select all users and their favorite product:

SELECT users.name AS user,  
products.name AS favorite  
FROM users **LEFT JOIN** products ON users.favorite\_product = products.id

Which will give you this result:

[  
  { user: 'John', favorite: 'Chocolate Heaven' },  
  { user: 'Peter', favorite: 'Chocolate Heaven' },  
  { user: 'Amy', favorite: 'Tasty Lemons' },  
  { user: 'Hannah', favorite: null },  
  { user: 'Michael', favorite: null }  
]

## Right Join

If you want to return all products, and the users who have them as their favorite, even if no user have them as their favorite, use the RIGHT JOIN statement:

### Example

Select all products and the user who have them as their favorite:

SELECT users.name AS user,  
products.name AS favorite  
FROM users **RIGHT JOIN** products ON users.favorite\_product = products.id

Which will give you this result:

[  
  { user: 'John', favorite: 'Chocolate Heaven' },  
  { user: 'Peter', favorite: 'Chocolate Heaven' },  
  { user: 'Amy', favorite: 'Tasty Lemons' },  
  { user: null, favorite: 'Vanilla Dreams' }  
]

**Note:** Hannah and Michael, who have no favorite product, are not included in the result.

# **Node.js MongoDB**

Node.js can be used in database applications.

One of the most popular NoSQL database is MongoDB.

## MongoDB

To be able to experiment with the code examples, you will need access to a MongoDB database.

You can download a free MongoDB database at [https://www.mongodb.com](https://www.mongodb.com/).

Or get started right away with a MongoDB cloud service at <https://www.mongodb.com/cloud/atlas>.

## Install MongoDB Driver

Let us try to access a MongoDB database with Node.js.

To download and install the official MongoDB driver, open the Command Terminal and execute the following:

Download and install mongodb package:

C:\Users\Your Name>npm install mongodb

Now you have downloaded and installed a mongodb database driver.

Node.js can use this module to manipulate MongoDB databases:

var mongo = require('mongodb');

# **Node.js MongoDB Create Database**

## Creating a Database

To create a database in MongoDB, start by creating a MongoClient object, then specify a connection URL with the correct ip address and the name of the database you want to create.

MongoDB will create the database if it does not exist, and make a connection to it.

### Example[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

Create a database called "mydb":

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/mydb";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  console.log("Database created!");  
  db.close();  
});

Save the code above in a file called "demo\_create\_mongo\_db.js" and run the file:

Run "demo\_create\_mongo\_db.js"

C:\Users\Your Name>node demo\_create\_mongo\_db.js

Which will give you this result:

Database created!

**Important:** In MongoDB, a database is not created until it gets content!

MongoDB waits until you have created a collection (table), with at least one document (record) before it actually creates the database (and collection).

# **Node.js MongoDB Create Collection**

A **collection** in MongoDB is the same as a **table** in MySQL

## Creating a Collection

To create a collection in MongoDB, use the createCollection() method:

### Example[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

Create a collection called "customers":

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
  dbo.createCollection("customers", function(err, res) {  
    if (err) throw err;  
    console.log("Collection created!");  
    db.close();  
  });  
});

Save the code above in a file called "demo\_mongodb\_createcollection.js" and run the file:

Run "demo\_mongodb\_createcollection.js"

C:\Users\Your Name>node demo\_mongodb\_createcollection.js

Which will give you this result:

Collection created!

**Important:** In MongoDB, a collection is not created until it gets content!

MongoDB waits until you have inserted a document before it actually creates the collection.

# **Node.js MongoDB Insert**

## Insert Into Collection

To insert a record, or document as it is called in MongoDB, into a collection, we use the insertOne() method.

A **document** in MongoDB is the same as a **record** in MySQL

The first parameter of the insertOne() method is an object containing the name(s) and value(s) of each field in the document you want to insert.

It also takes a callback function where you can work with any errors, or the result of the insertion:

### Example[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

Insert a document in the "customers" collection:

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
  var myobj = { name: "Company Inc", address: "Highway 37" };  
  dbo.collection("customers").insertOne(myobj, function(err, res) {  
    if (err) throw err;  
    console.log("1 document inserted");  
    db.close();  
  });  
});

Save the code above in a file called "demo\_mongodb\_insert.js" and run the file:

Run "demo\_mongodb\_insert.js"

C:\Users\Your Name>node demo\_mongodb\_insert.js

Which will give you this result:

1 document inserted

**Note:** If you try to insert documents in a collection that do not exist, MongoDB will create the collection automatically.

## Insert Multiple Documents

To insert multiple documents into a collection in MongoDB, we use the insertMany() method.

The first parameter of the insertMany() method is an array of objects, containing the data you want to insert.

It also takes a callback function where you can work with any errors, or the result of the insertion:

### Example

Insert multiple documents in the "customers" collection:

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
  var myobj = [  
    { name: 'John', address: 'Highway 71'},  
    { name: 'Peter', address: 'Lowstreet 4'},  
    { name: 'Amy', address: 'Apple st 652'},  
    { name: 'Hannah', address: 'Mountain 21'},  
    { name: 'Michael', address: 'Valley 345'},  
    { name: 'Sandy', address: 'Ocean blvd 2'},  
    { name: 'Betty', address: 'Green Grass 1'},  
    { name: 'Richard', address: 'Sky st 331'},  
    { name: 'Susan', address: 'One way 98'},  
    { name: 'Vicky', address: 'Yellow Garden 2'},  
    { name: 'Ben', address: 'Park Lane 38'},  
    { name: 'William', address: 'Central st 954'},  
    { name: 'Chuck', address: 'Main Road 989'},  
    { name: 'Viola', address: 'Sideway 1633'}  
  ];  
  dbo.collection("customers").insertMany(myobj, function(err, res) {  
    if (err) throw err;  
    console.log("Number of documents inserted: " + res.insertedCount);  
    db.close();  
  });  
});

Save the code above in a file called "demo\_mongodb\_insert\_multiple.js" and run the file:

Run "demo\_mongodb\_insert\_multiple.js"

C:\Users\Your Name>node demo\_mongodb\_insert\_multiple.js

Which will give you this result:

Number of documents inserted: 14

## The Result Object

When executing the insertMany() method, a result object is returned.

The result object contains information about how the insertion affected the database.

The object returned from the example above looked like this:

{  
  result: { ok: 1, n: 14 },  
  ops: [  
    { name: 'John', address: 'Highway 71', \_id: 58fdbf5c0ef8a50b4cdd9a84 },  
    { name: 'Peter', address: 'Lowstreet 4', \_id: 58fdbf5c0ef8a50b4cdd9a85 },  
    { name: 'Amy', address: 'Apple st 652', \_id: 58fdbf5c0ef8a50b4cdd9a86 },  
    { name: 'Hannah', address: 'Mountain 21', \_id: 58fdbf5c0ef8a50b4cdd9a87 },  
    { name: 'Michael', address: 'Valley 345', \_id: 58fdbf5c0ef8a50b4cdd9a88 },  
    { name: 'Sandy', address: 'Ocean blvd 2', \_id: 58fdbf5c0ef8a50b4cdd9a89 },  
    { name: 'Betty', address: 'Green Grass 1', \_id: 58fdbf5c0ef8a50b4cdd9a8a },  
    { name: 'Richard', address: 'Sky st 331', \_id: 58fdbf5c0ef8a50b4cdd9a8b },  
    { name: 'Susan', address: 'One way 98', \_id: 58fdbf5c0ef8a50b4cdd9a8c },  
    { name: 'Vicky', address: 'Yellow Garden 2', \_id: 58fdbf5c0ef8a50b4cdd9a8d },  
    { name: 'Ben', address: 'Park Lane 38', \_id: 58fdbf5c0ef8a50b4cdd9a8e },  
    { name: 'William', address: 'Central st 954', \_id: 58fdbf5c0ef8a50b4cdd9a8f },  
    { name: 'Chuck', address: 'Main Road 989', \_id: 58fdbf5c0ef8a50b4cdd9a90 },  
    { name: 'Viola', address: 'Sideway 1633', \_id: 58fdbf5c0ef8a50b4cdd9a91 } ],  
  insertedCount: 14,  
  insertedIds: [  
    58fdbf5c0ef8a50b4cdd9a84,  
    58fdbf5c0ef8a50b4cdd9a85,  
    58fdbf5c0ef8a50b4cdd9a86,  
    58fdbf5c0ef8a50b4cdd9a87,  
    58fdbf5c0ef8a50b4cdd9a88,  
    58fdbf5c0ef8a50b4cdd9a89,  
    58fdbf5c0ef8a50b4cdd9a8a,  
    58fdbf5c0ef8a50b4cdd9a8b,  
    58fdbf5c0ef8a50b4cdd9a8c,  
    58fdbf5c0ef8a50b4cdd9a8d,  
    58fdbf5c0ef8a50b4cdd9a8e,  
    58fdbf5c0ef8a50b4cdd9a8f  
    58fdbf5c0ef8a50b4cdd9a90,  
    58fdbf5c0ef8a50b4cdd9a91 ]  
}

The values of the properties can be displayed like this:

### Example

Return the number of inserted documents:

console.log(res.insertedCount)

Which will produce this result:

14

## The \_id Field

If you do not specify an \_id field, then MongoDB will add one for you and assign a unique id for each document.

In the example above no \_id field was specified, and as you can see from the result object, MongoDB assigned a unique \_id for each document.

If you do specify the \_id field, the value must be unique for each document:

### Example

Insert three records in a "products" table, with specified \_id fields:

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
  var myobj = [  
    { **\_id: 154**, name: 'Chocolate Heaven'},  
    { **\_id: 155**, name: 'Tasty Lemon'},  
    { **\_id: 156**, name: 'Vanilla Dream'}  
  ];  
  dbo.collection("products").insertMany(myobj, function(err, res) {  
    if (err) throw err;  
    console.log(res);  
    db.close();  
  });  
});

Save the code above in a file called "demo\_mongodb\_insert\_id.js" and run the file:

Run "demo\_mongodb\_insert\_id.js"

C:\Users\Your Name>node demo\_mongodb\_insert\_id.js

Which will give you this result:

{  
  result: { ok: 1, n: 3 },  
  ops: [  
    { \_id: 154, name: 'Chocolate Heaven },  
    { \_id: 155, name: 'Tasty Lemon },  
    { \_id: 156, name: 'Vanilla Dream } ],  
  insertedCount: 3,  
  insertedIds: [  
    154,  
    155,  
    156 ]  
}

# **Node.js MongoDB Find**

In MongoDB we use the **find** and **findOne** methods to find data in a collection.

Just like the **SELECT** statement is used to find data in a table in a MySQL database.

## Find One

To select data from a collection in MongoDB, we can use the findOne() method.

The findOne() method returns the first occurrence in the selection.

The first parameter of the findOne() method is a query object. In this example we use an empty query object, which selects all documents in a collection (but returns only the first document).

### Example[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

Find the first document in the customers collection:

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
  dbo.collection("customers").findOne({}, function(err, result) {  
    if (err) throw err;  
    console.log(result.name);  
    db.close();  
  });  
});

Save the code above in a file called "demo\_mongodb\_findone.js" and run the file:

Run "demo\_mongodb\_findone.js"

C:\Users\Your Name>node demo\_mongodb\_findone.js

Which will give you this result:

Company Inc.

## Find All

To select data from a table in MongoDB, we can also use the find() method.

The find() method returns all occurrences in the selection.

The first parameter of the find() method is a query object. In this example we use an empty query object, which selects all documents in the collection.

No parameters in the find() method gives you the same result as **SELECT \*** in MySQL.

### Example

Find all documents in the customers collection:

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
  dbo.collection("customers").find({}).toArray(function(err, result) {  
    if (err) throw err;  
    console.log(result);  
    db.close();  
  });  
});

Save the code above in a file called "demo\_mongodb\_find.js" and run the file:

Run "demo\_mongodb\_find.js"

C:\Users\Your Name>node demo\_mongodb\_find.js

Which will give you this result:

[  
  { \_id: 58fdbf5c0ef8a50b4cdd9a84 , name: 'John', address: 'Highway 71'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a85 , name: 'Peter', address: 'Lowstreet 4'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a86 , name: 'Amy', address: 'Apple st 652'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a87 , name: 'Hannah', address: 'Mountain 21'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a88 , name: 'Michael', address: 'Valley 345'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a89 , name: 'Sandy', address: 'Ocean blvd 2'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8a , name: 'Betty', address: 'Green Grass 1'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8b , name: 'Richard', address: 'Sky st 331'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8c , name: 'Susan', address: 'One way 98'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8d , name: 'Vicky', address: 'Yellow Garden 2'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8e , name: 'Ben', address: 'Park Lane 38'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8f , name: 'William', address: 'Central st 954'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a90 , name: 'Chuck', address: 'Main Road 989'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a91 , name: 'Viola', address: 'Sideway 1633'}  
]

## Find Some

The second parameter of the find() method is the projection object that describes which fields to include in the result.

This parameter is optional, and if omitted, all fields will be included in the result.

### Example

Return the fields "name" and "address" of all documents in the customers collection:

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
  dbo.collection("customers").find({},**{ projection: { \_id: 0, name: 1, address: 1 } }**).toArray(function(err, result) {  
    if (err) throw err;  
    console.log(result);  
    db.close();  
  });  
});

Save the code above in a file called "demo\_mongodb\_find\_fields.js" and run the file:

Run "demo\_mongodb\_find\_fields.js"

C:\Users\Your Name>node demo\_mongodb\_find\_fields.js

Which will give you this result:

[  
  { name: 'John', address: 'Highway 71'},  
  { name: 'Peter', address: 'Lowstreet 4'},  
  { name: 'Amy', address: 'Apple st 652'},  
  { name: 'Hannah', address: 'Mountain 21'},  
  { name: 'Michael', address: 'Valley 345'},  
  { name: 'Sandy', address: 'Ocean blvd 2'},  
  { name: 'Betty', address: 'Green Grass 1'},  
  { name: 'Richard', address: 'Sky st 331'},  
  { name: 'Susan', address: 'One way 98'},  
  { name: 'Vicky', address: 'Yellow Garden 2'},  
  { name: 'Ben', address: 'Park Lane 38'},  
  { name: 'William', address: 'Central st 954'},  
  { name: 'Chuck', address: 'Main Road 989'},  
  { name: 'Viola', address: 'Sideway 1633'}  
]

You are not allowed to specify both 0 and 1 values in the same object (except if one of the fields is the \_id field). If you specify a field with the value 0, all other fields get the value 1, and vice versa:

### Example

This example will exclude "address" from the result:

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
  dbo.collection("customers").find({},**{ projection: { address: 0 } }**).toArray(function(err, result) {  
    if (err) throw err;  
    console.log(result);  
    db.close();  
  });  
});

To exclude the \_id field, you must set its value to 0:

### Example

This example will return only the "name" field:

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
  dbo.collection("customers").find({},**{ projection: { \_id: 0, name: 1 } }**).toArray(function(err, result) {  
    if (err) throw err;  
    console.log(result);  
    db.close();  
  });  
});

### Example

This example will give you the same result as the first example; return all fields except the \_id field:

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
  dbo.collection("customers").find({},**{ projection: { \_id: 0 } }**).toArray(function(err, result) {  
    if (err) throw err;  
    console.log(result);  
    db.close();  
  });  
});

### Example

You get an error if you specify both 0 and 1 values in the same object (except if one of the fields is the \_id field):

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
  dbo.collection("customers").find({},**{ projection: { name: 1, address: 0 } }**).toArray(function(err, result) {  
    if (err) throw err;  
    console.log(result);  
    db.close();  
  });  
});

## The Result Object

As you can see from the result of the example above, the result can be converted into an array containing each document as an object.

To return e.g. the address of the third document, just refer to the third array object's address property:

### Example

Return the address of the third document:

console.log(result[2].address);

Which will produce this result:

Apple st 652

# **Node.js MongoDB Query**

## Filter the Result

When finding documents in a collection, you can filter the result by using a query object.

The first argument of the find() method is a query object, and is used to limit the search.

### Example[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

Find documents with the address "Park Lane 38":

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
**var query = { address: "Park Lane 38" };**  dbo.collection("customers").find(**query**).toArray(function(err, result) {  
    if (err) throw err;  
    console.log(result);  
    db.close();  
  });  
});

Save the code above in a file called "demo\_mongodb\_query.js" and run the file:

Run "demo\_mongodb\_query.js"

C:\Users\Your Name>node demo\_mongodb\_query.js

Which will give you this result:

[  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8e , name: 'Ben', address: 'Park Lane 38' }  
]

## Filter With Regular Expressions

You can write regular expressions to find exactly what you are searching for.

**Regular expressions can only be used to query strings.**

To find only the documents where the "address" field starts with the letter "S", use the regular expression /^S/:

### Example

Find documents where the address starts with the letter "S":

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
  var query = { address: **/^S/** };  
  dbo.collection("customers").find(query).toArray(function(err, result) {  
    if (err) throw err;  
    console.log(result);  
    db.close();  
  });  
});

Save the code above in a file called "demo\_mongodb\_query\_s.js" and run the file:

Run "demo\_mongodb\_query\_s.js"

C:\Users\Your Name>node demo\_mongodb\_query\_s.js

Which will give you this result:

[  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8b , name: 'Richard', address: 'Sky st 331' },  
  { \_id: 58fdbf5c0ef8a50b4cdd9a91 , name: 'Viola', address: 'Sideway 1633' }  
]

# **Node.js MongoDB Sort**

## Sort the Result

Use the sort() method to sort the result in ascending or descending order.

The sort() method takes one parameter, an object defining the sorting order.

### Example[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

Sort the result alphabetically by name:

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
**var mysort = { name: 1 };**  dbo.collection("customers").find()**.sort(mysort)**.toArray(function(err, result) {  
    if (err) throw err;  
    console.log(result);  
    db.close();  
  });  
});

Save the code above in a file called "demo\_sort.js" and run the file:

Run "demo\_sort.js"

C:\Users\Your Name>node demo\_sort.js

Which will give you this result:

[  
  { \_id: 58fdbf5c0ef8a50b4cdd9a86, name: 'Amy', address: 'Apple st 652'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8e, name: 'Ben', address: 'Park Lane 38'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8a, name: 'Betty', address: 'Green Grass 1'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a90, name: 'Chuck', address: 'Main Road 989'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a87, name: 'Hannah', address: 'Mountain 21'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a84, name: 'John', address: 'Highway 71'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a88, name: 'Michael', address: 'Valley 345'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a85, name: 'Peter', address: 'Lowstreet 4'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8b, name: 'Richard', address: 'Sky st 331'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a89, name: 'Sandy', address: 'Ocean blvd 2'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8c, name: 'Susan', address: 'One way 98'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8d, name: 'Vicky', address: 'Yellow Garden 2'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a91, name: 'Viola', address: 'Sideway 1633'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8f, name: 'William', address: 'Central st 954'}  
]

## Sort Descending

Use the value -1 in the sort object to sort descending.

{ name: 1 } // ascending  
{ name: -1 } // descending

### Example

Sort the result reverse alphabetically by name:

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
**var mysort = { name: -1 };**  dbo.collection("customers").find()**.sort(mysort)**.toArray(function(err, result) {  
    if (err) throw err;  
    console.log(result);  
    db.close();  
  });  
});

Save the code above in a file called "demo\_sort\_desc.js" and run the file:

Run "demo\_sort\_desc.js"

C:\Users\Your Name>node demo\_sort\_desc.js

Which will give you this result:

[  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8f, name: 'William', address: 'Central st 954'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a91, name: 'Viola', address: 'Sideway 1633'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8d, name: 'Vicky', address: 'Yellow Garden 2'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8c, name: 'Susan', address: 'One way 98'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a89, name: 'Sandy', address: 'Ocean blvd 2'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8b, name: 'Richard', address: 'Sky st 331'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a85, name: 'Peter', address: 'Lowstreet 4'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a88, name: 'Michael', address: 'Valley 345'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a84, name: 'John', address: 'Highway 71'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a87, name: 'Hannah', address: 'Mountain 21'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a90, name: 'Chuck', address: 'Main Road 989'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8a, name: 'Betty', address: 'Green Grass 1'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8e, name: 'Ben', address: 'Park Lane 38'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a86, name: 'Amy', address: 'Apple st 652'}  
]

# **Node.js MongoDB Delete**

## Delete Document

To delete a record, or document as it is called in MongoDB, we use the deleteOne() method.

The first parameter of the deleteOne() method is a query object defining which document to delete.

**Note:** If the query finds more than one document, only the first occurrence is deleted.

### Example[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

Delete the document with the address "Mountain 21":

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
**var myquery = { address: 'Mountain 21' };**  dbo.collection("customers").deleteOne(myquery, function(err, obj) {  
    if (err) throw err;  
    console.log("1 document deleted");  
    db.close();  
  });  
});

Save the code above in a file called "demo\_delete.js" and run the file:

Run "demo\_delete.js"

C:\Users\Your Name>node demo\_delete.js

Which will give you this result:

1 document deleted

## Delete Many

To delete more than one document, use the deleteMany() method.

The first parameter of the deleteMany() method is a query object defining which documents to delete.

### Example

Delete all documents were the address starts with the letter "O":

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
**var myquery = { address: /^O/ };**  dbo.collection("customers").deleteMany(myquery, function(err, obj) {  
    if (err) throw err;  
    console.log(obj.result.n + " document(s) deleted");  
    db.close();  
  });  
});

Save the code above in a file called "demo\_delete\_many.js" and run the file:

Run "demo\_delete\_many.js"

C:\Users\Your Name>node demo\_delete\_many.js

Which will give you this result:

2 document(s) deleted

## The Result Object

The deleteMany() method returns an object which contains information about how the execution affected the database.

Most of the information is not important to understand, but one object inside the object is called "result" which tells us if the execution went OK, and how many documents were affected.

The result object looks like this:

{ n: 2, ok: 1 }

You can use this object to return the number of deleted documents:

### Example

Return the number of deleted documents:

console.log(obj.result.n);

Which will produce this result:

2

# **Node.js MongoDB Drop**

## Drop Collection

You can delete a table, or collection as it is called in MongoDB, by using the drop() method.

The drop() method takes a callback function containing the error object and the result parameter which returns true if the collection was dropped successfully, otherwise it returns false.

### Example[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

Delete the "customers" table:

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
  dbo.collection("customers").drop(function(err, delOK) {  
    if (err) throw err;  
    if (delOK) console.log("Collection deleted");  
    db.close();  
  });  
});

Save the code above in a file called "demo\_drop.js" and run the file:

Run "demo\_drop.js"

C:\Users\Your Name>node demo\_drop.js

Which will give you this result:

Collection deleted

## db.dropCollection

You can also use the dropCollection() method to delete a table (collection).

The dropCollection() method takes two parameters: the name of the collection and a callback function.

### Example

Delete the "customers" collection, using dropCollection():

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb"); **dbo.dropCollection("customers", function(err, delOK) {**    if (err) throw err;  
    if (delOK) console.log("Collection deleted");  
    db.close();  
  });  
});

Save the code above in a file called "demo\_dropcollection.js" and run the file:

Run "demo\_dropcollection.js"

C:\Users\Your Name>node demo\_dropcollection.js

Which will give you this result:

Collection deleted

# **Node.js MongoDB Update**

## Update Document

You can update a record, or document as it is called in MongoDB, by using the updateOne() method.

The first parameter of the updateOne() method is a query object defining which document to update.

**Note:** If the query finds more than one record, only the first occurrence is updated.

The second parameter is an object defining the new values of the document.

### Example[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

Update the document with the address "Valley 345" to name="Mickey" and address="Canyon 123":

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://127.0.0.1:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
  var myquery = { address: "Valley 345" };  
  var newvalues = { $set: {name: "Mickey", address: "Canyon 123" } };  
  dbo.collection("customers").updateOne(myquery, newvalues, function(err, res) {  
    if (err) throw err;  
    console.log("1 document updated");  
    db.close();  
  });  
});

Save the code above in a file called "demo\_update\_one.js" and run the file:

Run "demo\_update\_one.js"

C:\Users\Your Name>node demo\_update\_one.js

Which will give you this result:

1 document updated

## Update Only Specific Fields

When using the $set operator, only the specified fields are updated:

### Example

Update the address from "Valley 345" to "Canyon 123":

...  
  var myquery = { address: "Valley 345" };  
  var newvalues = { **$set: { address: "Canyon 123" }** };  
  dbo.collection("customers").updateOne(myquery, newvalues, function(err, res) {  
...

## Update Many Documents

To update all documents that meets the criteria of the query, use the updateMany() method.

### Example

Update all documents where the name starts with the letter "S":

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://127.0.0.1:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
  var myquery = { address: /^S/ };  
  var newvalues = {$set: {name: "Minnie"} };  
  dbo.collection("customers").updateMany(myquery, newvalues, function(err, res) {  
    if (err) throw err;  
    console.log(res.result.nModified + " document(s) updated");  
    db.close();  
  });  
});

Save the code above in a file called "demo\_update\_many.js" and run the file:

Run "demo\_update\_many.js"

C:\Users\Your Name>node demo\_update\_many.js

Which will give you this result:

2 document(s) updated

## The Result Object

The updateOne() and the updateMany() methods return an object which contains information about how the execution affected the database.

Most of the information is not important to understand, but one object inside the object is called "result" which tells us if the execution went OK, and how many documents were affected.

The result object looks like this:

{ n: 1, nModified: 2, ok: 1 }

You can use this object to return the number of updated documents:

### Example

Return the number of updated documents:

console.log(res.result.nModified);

Which will produce this result:

2

# **Node.js MongoDB Limit**

## Limit the Result

To limit the result in MongoDB, we use the limit() method.

The limit() method takes one parameter, a number defining how many documents to return.

Consider you have a "customers" collection:

### customers[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

[  
  { \_id: 58fdbf5c0ef8a50b4cdd9a84 , name: 'John', address: 'Highway 71'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a85 , name: 'Peter', address: 'Lowstreet 4'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a86 , name: 'Amy', address: 'Apple st 652'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a87 , name: 'Hannah', address: 'Mountain 21'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a88 , name: 'Michael', address: 'Valley 345'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a89 , name: 'Sandy', address: 'Ocean blvd 2'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8a , name: 'Betty', address: 'Green Grass 1'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8b , name: 'Richard', address: 'Sky st 331'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8c , name: 'Susan', address: 'One way 98'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8d , name: 'Vicky', address: 'Yellow Garden 2'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8e , name: 'Ben', address: 'Park Lane 38'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a8f , name: 'William', address: 'Central st 954'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a90 , name: 'Chuck', address: 'Main Road 989'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a91 , name: 'Viola', address: 'Sideway 1633'}  
]

### Example

Limit the result to only return 5 documents:

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
  dbo.collection("customers").find()**.limit(5)**.toArray(function(err, result) {  
    if (err) throw err;  
    console.log(result);  
    db.close();  
  });  
});

Save the code above in a file called "demo\_mongodb\_limit.js" and run the file:

Run "demo\_mongodb\_limit.js"

C:\Users\Your Name>node demo\_mongodb\_limit.js

Which will give you this result:

### customers

[  
  { \_id: 58fdbf5c0ef8a50b4cdd9a84 , name: 'John', address: 'Highway 71'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a85 , name: 'Peter', address: 'Lowstreet 4'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a86 , name: 'Amy', address: 'Apple st 652'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a87 , name: 'Hannah', address: 'Mountain 21'},  
  { \_id: 58fdbf5c0ef8a50b4cdd9a88 , name: 'Michael', address: 'Valley 345'}  
]

As you can see from the result above, only the 5 first documents were returned.

# **Node.js MongoDB Join**

## Join Collections

MongoDB is not a relational database, but you can perform a left outer join by using the $lookup stage.

The $lookup stage lets you specify which collection you want to join with the current collection, and which fields that should match.

Consider you have a "orders" collection and a "products" collection:

### orders[Get your own Node.js Server](https://www.w3schools.com/nodejs/nodejs_server.asp)

[  
  { \_id: 1, product\_id: 154, status: 1 }  
]

### products

[  
  { \_id: 154, name: 'Chocolate Heaven' },  
  { \_id: 155, name: 'Tasty Lemons' },  
  { \_id: 156, name: 'Vanilla Dreams' }  
]

### Example

Join the matching "products" document(s) to the "orders" collection:

var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://127.0.0.1:27017/";  
  
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  var dbo = db.db("mydb");  
  dbo.collection('orders').aggregate([  
**{ $lookup:  
       {  
         from: 'products',  
         localField: 'product\_id',  
         foreignField: '\_id',  
         as: 'orderdetails'  
       }  
     }**    ]).toArray(function(err, res) {  
    if (err) throw err;  
    console.log(JSON.stringify(res));  
    db.close();  
  });  
});

Save the code above in a file called "demo\_mongodb\_join.js" and run the file:

Run "demo\_mongodb\_join.js"

C:\Users\Your Name>node demo\_mongodb\_join.js

Which will give you this result:

[  
  { "\_id": 1, "product\_id": 154, "status": 1, "orderdetails": [  
    { "\_id": 154, "name": "Chocolate Heaven" } ]  
  }  
]

As you can see from the result above, the matching document from the products collection is included in the orders collection as an array.

THANK YOU