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

Download Node.js

The official Node.js website has installation instructions for Node.js: https://nodejs.org

After install node js software and vs code .

Open VS 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);
```

open terminal and

write `node myfirst.js` and hit enter:

Start your internet browser, and type in the address: http://localhost:8080

Chapter 2

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

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

Chapter 3

Node.js HTTP Module

he 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

```
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

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 chapter.

Chapter 4

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

```
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

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!');
});
```

chapter 5

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'
```

Run example »

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!

Chapter 6

What is NPM?

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

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

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

```
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

Chapter 7

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

```
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');
```

Chapter 8

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

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);
```

Chapter 9

Database connection with Mysql

Node.js MySQL

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);
  });
});
```

Insert Into Table

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

Example

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
```

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
```

Select Statement

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
```

Selecting From a Table

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

Example

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

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);
});
```

Node.js MySQL Order By

```
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", function (err,
result) {
```

```
    if (err) throw err;
    console.log(result);
  });
});
```

```
C:\Users\          >node demo_db_orderby.js
```

Order by desc

```
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);
  });
});
```

```
C:\Users\          >node demo_db_orderby_desc.js
```

Delete Record

```
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);
  });
});
```

```
C:\Users\          >node demo_db_delete.js
```

Update Table

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

Example

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");
  });
});
```

```
C:\Users\          >node demo_db_update.js
```

Limit the Result

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

Example

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);
  });
});
```

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

```
[
  { 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' }
]
```

Install new MONGODB new version 6.0

Native MongoDB driver for Node.js

The native MongoDB driver for Node.JS is a dependency that allows our JavaScript application to interact with the NoSQL database, either locally or on the cloud through MongoDB Atlas. We are allowed to use promises as well as callbacks that gives us greater flexibility in using ES6 features.

In order to start working with the MongoDB driver, we shall first create a new folder and initialize our project:

```
npm init -y
```

Here, -y is a flag which will initialize our project with default values.

We will install the MongoDB driver and save it as a dependency with the following command:

```
npm install mongodb --save
```

In our JavaScript entry point file, for the sake of convenience, we shall name *app.js* and we will write the following code to connect to the server:

- JavaScript

```
// Importing MongoClient from mongodb driver

const { MongoClient } = require('mongodb');



// Connecting to a local port

const uri = 'mongodb://127.0.0.1:27017';



const client = new MongoClient(uri, {

    useUnifiedTopology: true,

    useNewUrlParser: true

});



connect();



// ESNext syntax using async-await

async function connect() {

    try {
```

```
        await client.connect();

        const db = client.db('cars');

        console.log(

    `Successfully connected to db ${db.databaseName}`);

    }

    catch (err) {

        console.error(`we encountered ${err}`);

    }

    finally {

        client.close();

    }

}
```

Output:

```
Successfully connected to db cars
```

Now that we have made the connection, let us see some basic Insertion, Read, Update and Delete Operations on our database:
Insertion and Read: In the following code snippet we are going to deal with Insertion and Read operation.

- JavaScript

```
const { MongoClient } = require('mongodb');

const uri = 'mongodb://127.0.0.1:27017';



const client = new MongoClient(uri, {
```

```
    useUnifiedTopology: true,
    useNewUrlParser: true
});

connect();

async function connect() {
    try {
        await client.connect();
        const db = client.db('cars');
        console.log(
    `Successfully connected to db ${db.databaseName}`);

        const sportsCars = db.collection('SportsCars');

        // Insertion
        const cursorInsertion = await sportsCars.insertMany([
            {
                'company': 'mercedes',
                'series': 'Black Series',
                'model': 'SLS AMG'
            },
```

```
            {
                'company': 'Audi',
                'series': 'A series',
                'model': 'A8'
            }]);
        console.log(cursorInsertion.insertedCount);


        // Display
        const cursorFind = sportsCars.find();
        const data = await cursorFind.toArray();
        console.table(data);
    }
    catch (err) {
        console.error(`we encountered ${err}`);
    }
    finally {
        client.close();
    }
}
```

Explanation: A collection called sports cars is created using the collections() method. For Insertion, we use the two following methods:

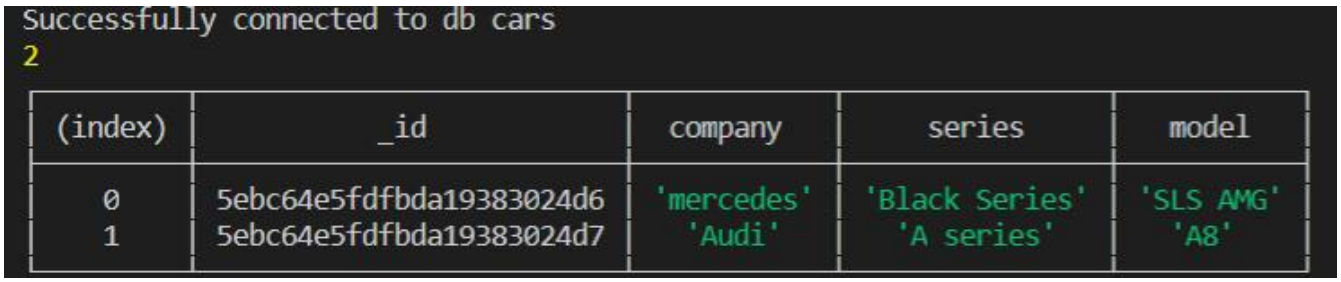
1. ***insertMany() method:*** This method is used to insert more than one entry into the database with the help of cursors. In this case, it takes an array of objects as parameters. The method returns a promise, hence we used the

await keyword. Alternatively, the method insertOne() is used to insert a single document into the table.
2. ***InsertedCount:*** This function is used to count the number of insertions that were made.

For Display we used the following methods:

1. ***find():*** This method is used to find all the documents in the database with the help of cursors.
2. ***toArray():*** This method uses the cursor element received from the find() method to store the database in an array of objects. This method returns a promise, hence we have used the keyword await.

The Output of the Snippet is as follows:



Update: The following code snippet will help us to update a database element and then we shall display the updated database:

- JavaScript

```
const { MongoClient } = require('mongodb');

const uri = 'mongodb://127.0.0.1:27017';



const client = new MongoClient(uri, {

    useUnifiedTopology: true,

    useNewUrlParser: true

});



connect();
```

```
async function connect() {
    try {
        await client.connect();
        const db = client.db('cars');

        const sportsCars = db.collection('SportsCars');
        //update
        const cursorUpdate = await sportsCars.updateOne(
            { "company": "mercedes" },
            { "$set": { "status": "sold" } }
        );

        console.log(cursorUpdate.modifiedCount);

        // Display
        const cursorFind = sportsCars.find();
        const data = await cursorFind.toArray();
        console.table(data);
    }
    catch (err) {
        console.error(`we encountered ${err}`);
```

```
    }

    finally {

        client.close();

    }

}
```

Explanation: We use the following methods for updating the database:

1. ***updateOne() method:*** This methods allows us to update one entry. The first argument it takes is a key-value pair corresponding to the database entry as we want to update. It can be any of the properties that the element possesses. The second argument is an update command $set, which is paired with an object. The object is again a key-value pair of either an existing or a new property. If the property already exists, then the property is updated with the value passed. If it does not exist, then it is added. This method returns a promise, hence we use the keyword await. Alternatively, updateMany() method can be used to update multiple documents.
2. ***modifiedCount:*** This method is called on the cursor element received from the previous method and gives us a count of the number of entries updated.

Output:

```
1
```

(index)	_id	company	series	model	status
0	5ebc6c6112796124444f7154	'mercedes'	'Black Series'	'SLS AMG'	'sold'
1	5ebc6c6112796124444f7155	'Audi'	'A series'	'A8'	

Deleting an Entry: In the following snippet, we will delete an entry based on series:

- JavaScript

```
const { MongoClient } = require('mongodb');

const uri = 'mongodb://127.0.0.1:27017';
```

```
const client = new MongoClient(uri, {
    useUnifiedTopology: true,
    useNewUrlParser: true
});

connect();

async function connect() {
    try {
        await client.connect();
        const db = client.db('cars');

        const sportsCars = db.collection('SportsCars');
        //Delete
        const cursorDelete = await sportsCars
                .deleteOne({ "series": "A series" });

        console.log(cursorDelete.deletedCount);

        // Display
```

```
        const cursorFind = sportsCars.find();

        const data = await cursorFind.toArray();

        console.table(data);

    }

    catch (err) {

        console.error(`we encountered ${err}`);

    }

    finally {

        client.close();

    }

}
```

Explanation: We use the following methods to delete the entry “company” : “Audi” from the database:

1. ***deleteOne:*** This method is used to delete one entry from the database. It takes in a key-value pair which corresponds to the entry that we want to delete. This method returns a promise. Hence, we use the keyword await. Alternatively, deleteMany() method can be used to delete multiple documents at once.
2. ***deletedCount:*** This method is called on the cursor element received from the previous method and it returns the number of deletions that were made.

Output:

```
1
```

(index)	_id	company	series	model	status
0	5ebc6c6112796124444f7154	'mercedes'	'Black Series'	'SLS AMG'	'sold'

Hence, Node and MongoDB can be easily used to make efficient backend CRUD apis.

Express Chapter

Express Overview

Express is a minimal and flexible Node.js web application framework that provides a robust set of features to develop web and mobile applications. It facilitates the rapid development of Node based Web applications. Following are some of the core features of Express framework −

- Allows to set up middlewares to respond to HTTP Requests.
- Defines a routing table which is used to perform different actions based on HTTP Method and URL.
- Allows to dynamically render HTML Pages based on passing arguments to templates.

Installing Express

Firstly, install the Express framework globally using NPM so that it can be used to create a web application using node terminal.

```
$ npm install express --save
```

The above command saves the installation locally in the **node_modules** directory and creates a directory express inside node_modules. You should install the following important modules along with express −

- **body-parser** − This is a node.js middleware for handling JSON, Raw, Text and URL encoded form data.
- **cookie-parser** − Parse Cookie header and populate req.cookies with an object keyed by the cookie names.
- **multer** − This is a node.js middleware for handling multipart/form-data.

```
npm install body-parser --save
npm install cookie-parser --save
npm install multer --save
```

Hello world Example

Following is a very basic Express app which starts a server and listens on port 8081 for connection. This app responds with **Hello World!** for requests to the homepage. For every other path, it will respond with a **404 Not Found.**

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello World');
})

var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port
```

```
  console.log("Example app listening at http://%s:%s", host, port)
})
```

Save the above code in a file named server.js and run it with the following command.

```
node server.js
```

You will see the following output −

```
Example app listening at http://0.0.0.0:8081
```

Open http://127.0.0.1:8081/ in any browser to see the following result.

Request & Response

Express application uses a callback function whose parameters are **request** and **response** objects.

```
app.get('/', function (req, res) {
  // --
})
```

- Request Object − The request object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on.
- Response Object − The response object represents the HTTP response that an Express app sends when it gets an HTTP request.

You can print **req** and **res** objects which provide a lot of information related to HTTP request and response including cookies, sessions, URL, etc.

Basic Routing

We have seen a basic application which serves HTTP request for the homepage. Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).

We will extend our Hello World program to handle more types of HTTP requests.

```
var express = require('express');
var app = express();

// This responds with "Hello World" on the homepage
app.get('/', function (req, res) {
  console.log("Got a GET request for the homepage");
  res.send('Hello GET');
})

// This responds a POST request for the homepage
app.post('/', function (req, res) {
  console.log("Got a POST request for the homepage");
  res.send('Hello POST');
})

// This responds a DELETE request for the /del_user page.
app.delete('/del_user', function (req, res) {
  console.log("Got a DELETE request for /del_user");
  res.send('Hello DELETE');
})

// This responds a GET request for the /list_user page.
app.get('/list_user', function (req, res) {
  console.log("Got a GET request for /list_user");
  res.send('Page Listing');
})

// This responds a GET request for abcd, abxcd, ab123cd, and so on
app.get('/ab*cd', function(req, res) {
  console.log("Got a GET request for /ab*cd");
  res.send('Page Pattern Match');
})

var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)
})
```

Save the above code in a file named server.js and run it with the following command.

```
$ node server.js
```

You will see the following output −

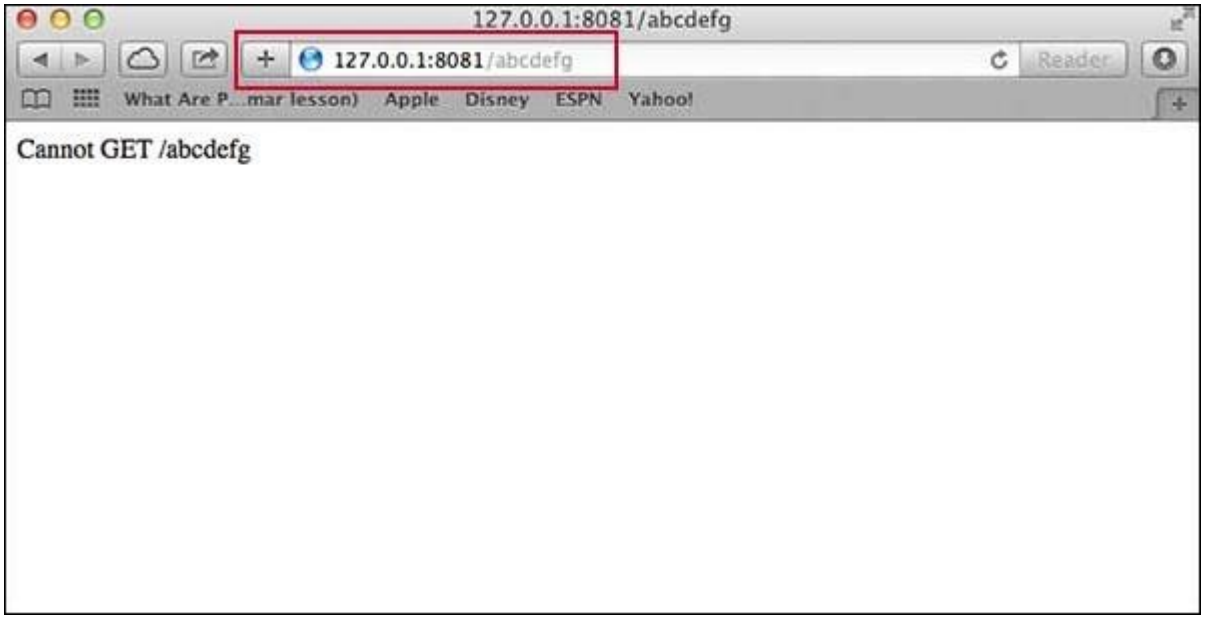
```
Example app listening at http://0.0.0.0:8081
```

Now you can try different requests at http://127.0.0.1:8081 to see the output generated by server.js. Following are a few screens shots showing different responses for different URLs.

Screen showing again http://127.0.0.1:8081/list_user

Screen showing again http://127.0.0.1:8081/abcd

Screen showing again http://127.0.0.1:8081/abcdefg



Serving Static Files

Express provides a built-in middleware **express.static** to serve static files, such as images, CSS, JavaScript, etc.

You simply need to pass the name of the directory where you keep your static assets, to the **express.static** middleware to start serving the files directly. For example, if you keep your images, CSS, and JavaScript files in a directory named public, you can do this −

```
app.use(express.static('public'));
```

We will keep a few images in **public/images** sub-directory as follows −

```
node_modules
server.js
public/
public/images
public/images/logo.png
```

Let's modify "Hello Word" app to add the functionality to handle static files.

```
var express = require('express');
var app = express();

app.use(express.static('public'));

app.get('/', function (req, res) {
  res.send('Hello World');
})

var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)
})
```

Save the above code in a file named server.js and run it with the following command.

```
$ node server.js
```

Now open http://127.0.0.1:8081/images/logo.png in any browser and see observe following result.

GET Method

Here is a simple example which passes two values using HTML FORM GET method. We are going to use **process_get** router inside server.js to handle this input.

```
<html>
  <body>

    <form action = "http://127.0.0.1:8081/process_get" method = "GET">
      First Name: <input type = "text" name = "first_name">  <br>
      Last Name: <input type = "text" name = "last_name">
      <input type = "submit" value = "Submit">
    </form>

  </body>
</html>
```

Let's save above code in index.htm and modify server.js to handle home page requests as well as the input sent by the HTML form.

```
var express = require('express');
var app = express();

app.use(express.static('public'));
app.get('/index.htm', function (req, res) {
  res.sendFile( __dirname + "/" + "index.htm" );
})

app.get('/process_get', function (req, res) {
  // Prepare output in JSON format
  response = {
    first_name:req.query.first_name,
    last_name:req.query.last_name
  };
  console.log(response);
  res.end(JSON.stringify(response));
})

var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)
})
```

Accessing the HTML document using *http://127.0.0.1:8081/index.htm* will generate the following form −

First Name:	
Last Name:	

Now you can enter the First and Last Name and then click submit button to see the result and it should return the following result −

```
{"first_name":"John","last_name":"Paul"}
```

POST Method

Here is a simple example which passes two values using HTML FORM POST method. We are going to use **process_get** router inside server.js to handle this input.

```
<html>
  <body>

    <form action = "http://127.0.0.1:8081/process_post" method = "POST">
      First Name: <input type = "text" name = "first_name"> <br>
      Last Name: <input type = "text" name = "last_name">
      <input type = "submit" value = "Submit">
    </form>

  </body>
</html>
```

Let's save the above code in index.htm and modify server.js to handle home page requests as well as the input sent by the HTML form.

```
var express = require('express');
var app = express();
var bodyParser = require('body-parser');

// Create application/x-www-form-urlencoded parser
var urlencodedParser = bodyParser.urlencoded({ extended: false })

app.use(express.static('public'));
app.get('/index.htm', function (req, res) {
  res.sendFile( __dirname + "/" + "index.htm" );
})

app.post('/process_post', urlencodedParser, function (req, res) {
  // Prepare output in JSON format
  response = {
    first_name:req.body.first_name,
    last_name:req.body.last_name
  };
```

```
  console.log(response);
  res.end(JSON.stringify(response));
})

var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)
})
```

Accessing the HTML document using *http://127.0.0.1:8081/index.html* will generate the following form −

First Name:	
Last Name:	

Now you can enter the First and Last Name and then click the submit button to see the following result −

```
{"first_name":"John","last_name":"Paul"}
```

What is REST architecture?

REST stands for REpresentational State Transfer. REST is web standards based architecture and uses HTTP Protocol. It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods. REST was first introduced by Roy Fielding in 2000.

A REST Server simply provides access to resources and REST client accesses and modifies the resources using HTTP protocol. Here each resource is identified by URIs/ global IDs. REST uses various representation to represent a resource like text, JSON, XML but JSON is the most popular one.

HTTP methods

Following four HTTP methods are commonly used in REST based architecture.

- **GET** − This is used to provide a read only access to a resource.
- **PUT** − This is used to create a new resource.
- **DELETE** − This is used to remove a resource.
- **POST** − This is used to update a existing resource or create a new resource.

RESTful Web Services

A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to

exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., communication between Java and Python, or Windows and Linux applications) is due to the use of open standards.

Web services based on REST Architecture are known as RESTful web services. These webservices uses HTTP methods to implement the concept of REST architecture. A RESTful web service usually defines a URI, Uniform Resource Identifier a service, which provides resource representation such as JSON and set of HTTP Methods.

RESTFUL

Creating RESTful for A Library

Consider we have a JSON based database of users having the following users in a file **users.json**:

```
{
  "user1" : {
    "name" : "mahesh",
    "password" : "password1",
    "profession" : "teacher",
    "id": 1
  },

  "user2" : {
    "name" : "suresh",
    "password" : "password2",
    "profession" : "librarian",
    "id": 2
  },

  "user3" : {
    "name" : "ramesh",
    "password" : "password3",
    "profession" : "clerk",
    "id": 3
  }
}
```

Based on this information we are going to provide following RESTful APIs.

Sr.No.	URI	HTTP Method	POST body	Result
1	listUsers	GET	empty	Show list of all the users.

2	addUser	POST	JSON String	Add details of new user.
3	deleteUser	DELETE	JSON String	Delete an existing user.
4	:id	GET	empty	Show details of a user.

I'm keeping most of the part of all the examples in the form of hard coding assuming you already know how to pass values from front end using Ajax or simple form data and how to process them using express **Request** object.

List Users

Let's implement our first RESTful API **listUsers** using the following code in a server.js file −

server.js

```
var express = require('express');
var app = express();
var fs = require("fs");

app.get('/listUsers', function (req, res) {
  fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
    console.log( data );
    res.end( data );
  });
})

var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port
  console.log("Example app listening at http://%s:%s", host, port)
})
```

Now try to access defined API using *URL: http://127.0.0.1:8081/listUsers* and *HTTP Method : GET* on local machine using any REST client. This should produce following result −

You can change given IP address when you will put the solution in production environment.

```
{
  "user1" : {
    "name" : "mahesh",
    "password" : "password1",
    "profession" : "teacher",
    "id": 1
  },

  "user2" : {
    "name" : "suresh",
```

```
    "password" : "password2",
    "profession" : "librarian",
    "id": 2
  },

  "user3" : {
    "name" : "ramesh",
    "password" : "password3",
    "profession" : "clerk",
    "id": 3
  }
}
```

Add User

Following API will show you how to add new user in the list. Following is the detail of the new user −

```
user = {
  "user4" : {
    "name" : "mohit",
    "password" : "password4",
    "profession" : "teacher",
    "id": 4
  }
}
```

You can accept the same input in the form of JSON using Ajax call but for teaching point of view, we are making it hard coded here. Following is the **addUser** API to a new user in the database −

server.js

```
var express = require('express');
var app = express();
var fs = require("fs");

var user = {
  "user4" : {
    "name" : "mohit",
    "password" : "password4",
    "profession" : "teacher",
    "id": 4
  }
}

app.get('/addUser', function (req, res) {
  // First read existing users.
  fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
    data = JSON.parse( data );
    data["user4"] = user["user4"];
    console.log( data );
    res.end( JSON.stringify(data));
  });
```

```
})

var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port
  console.log("Example app listening at http://%s:%s", host, port)
})
```

Now try to access defined API using *URL: http://127.0.0.1:8081/addUser* and *HTTP Method : POST* on local machine using any REST client. This should produce following result −

```
{
  "user1":{"name":"mahesh","password":"password1","profession":"teacher","id":1},
  "user2":{"name":"suresh","password":"password2","profession":"librarian","id":2},
  "user3":{"name":"ramesh","password":"password3","profession":"clerk","id":3},
  "user4":{"name":"mohit","password":"password4","profession":"teacher","id":4}
}
```

Show Detail

Now we will implement an API which will be called using user ID and it will display the detail of the corresponding user.

server.js

```
var express = require('express');
var app = express();
var fs = require("fs");

app.get('/:id', function (req, res) {
  // First read existing users.
  fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
     var users = JSON.parse( data );
     var user = users["user" + req.params.id]
     console.log( user );
     res.end( JSON.stringify(user));
  });
})

var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port
  console.log("Example app listening at http://%s:%s", host, port)
})
```

Now try to access defined API using *URL: http://127.0.0.1:8081/2* and *HTTP Method : GET* on local machine using any REST client. This should produce following result −

```
{"name":"suresh","password":"password2","profession":"librarian","id":2}
```

Delete User

This API is very similar to addUser API where we receive input data through req.body and then based on user ID we delete that user from the database. To keep our program simple we assume we are going to delete user with ID 2.

server.js

```
var express = require('express');
var app = express();
var fs = require("fs");

var id=2
app.get('/deleteUser/:id', function (req, res) {
   // First read existing users.
   fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
      data = JSON.parse(data);
      delete data["user"+2];

      console.log(data);
      res.end( JSON.stringify(data));
   });
})

var server = app.listen(8081, function () {
   var host = server.address().address
   var port = server.address().port
   console.log("Example app listening at http://%s:%s", host, port)
})
```

Now try to access defined API using *URL: http://127.0.0.1:8081/deleteUser* and *HTTP Method : DELETE* on local machine using any REST client. This should produce following result −

{"user1":{"name":"mahesh","password":"password1","profession":"teacher","id":1},
"user3":{"name":"ramesh","password":"password3","profession":"clerk","id":3

Insert Data in form format and connection with mysql and node js

Database -julkar

Table name- info

id int auto increment

name varchar(50)

email varchar(40)

city varchar(40)

Open vs code and create folder node_js and then open terminal and type below command

```
PS C:\Users\admin\Desktop\node_js> mkdir myapp

PS C:\Users\admin\Desktop\node_js> cd myapp

PS C:\Users\admin\Desktop\node_js\myapp> npm init -y
```

```
{
  "name": "myapp",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

```
PS C:\Users\admin\Desktop\node_js\myapp> npm install express --save
PS C:\Users\admin\Desktop\node_js\myapp> npm install body-parser --save
PS C:\Users\admin\Desktop\node_js\myapp> npm install ejs --save
PS C:\Users\admin\Desktop\node_js\myapp> npm install mysql --save
```

Look Folder structure in vs code

Node_js

-Myapp

-node_module

-views

-forms.html

-add.js

-Package.json

-package-lock.json

Forms.html

```
<html>
<head>
<title>Personal Information</title>
</head>
<body>
<div id="info">
<h1>Personal Information</h1>
<form action="/add" method="post">

    <label for="name">Name:</label>
    <input type="text" id="name" name="name"
placeholder="Enter your full name" />
  <br><br>
    <label for="email">Email:</label>
    <input type="email" id="email" name="email"
placeholder="Enter your email address" />
  <br><br>
    <label for="city">City:</label>
    <input type="text" id="city" name="city" placeholder="Enter
your city" />
```

```
   <br><br>
   <label for="pincode">Pincode:</label>
      <input type="text" id="pincode" name="pincode"
placeholder="Enter your pincode" />
   <br><br>
      <input type="submit" value="Send message" />
</form>
</div>
</body>
</html>
```

Add.js

```
var express = require('express');
var app = express();
var ejs = require('ejs');
var mysql = require('mysql');
var bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({ extended: true }));

var con = mysql.createConnection({
   host: "localhost",
```

```
    user: "root",
    password: "",
    database: "julkar"
  });
app.get('/',function(req,res,next){
    res.sendfile('views/forms.html');
});
app.post('/add', function(req, res)
{
console.log('req.body');
console.log(req.body);
res.write('You sent the name "' + req.body.name+'".\n');
res.write('You sent the Email "' + req.body.email+'".\n');
res.write('You sent the City "' + req.body.city+'".\n');
res.write('You sent the Pincode "' + req.body.pincode+'".\n');
res.end()

con.connect(function(err) {
    if (err) throw err;
    var sql = "INSERT INTO info (name, email,city,pincode) VALUES
('"+req.body.name+"', '"+
req.body.email+"','"+req.body.city+"','"+req.body.pincode+"')";
```

```
    con.query(sql, function (err, result) {
      if (err) throw err;
      console.log("1 record inserted");
       res.end();
    });
    });
});
app.listen(3000);
console.log('Example app listening at port:3000');
```

Type **node add.js** in terminal

Open Browser

Localhost:3000

Now

Node_js

-Myapp

-node_module

-views

-forms.html

-delete.html

-update.html

-add.js

-delete.js

-update.js

-Package.json

-package-lock.json

Delete.html

```
<html>
   <head>
      <title>Personal Information</title>
   </head>
   <body>
      <div id="info">
         <h1>Personal Information</h1>
         <form action="/delete" method="post">
            <label for="id">
               ID:
```

```
            </label>
            <input type="text" id="id" name="id" placeholder="enter
your ID">
            <input type="submit" value="Delete">
        </form>
    </div>
  </body>
</html>
```

Update.html

```
<html>
  <head>
    <title>Personal Information</title>
  </head>
  <body>
    <div id="info">
      <h1>Personal Information</h1>
      <form action="/update" method="post">
        <label for="id">
          ID:
        </label>
```

```
        <input type="text" id="id" name="id" placeholder="enter your ID">
        <br><br>
        <label for="name">
            Name:
        </label>
        <input type="text" id="name" name="name" placeholder="enter your full name">
        <br><br>
        <label for="email">
            Email:
        </label>
        <input type="email" id="email" name="email" placeholder="enter your email address">
        <br><br>
        <label for="city">
            City:
        </label>
        <input type="text" id="city" name="city" placeholder="enter your city">
        <br><br>
        <label for="pincode">
            Pincode:
```

```
            </label>
            <input type="text" id="pincode" name="pincode"
placeholder="enter your pincode">
            <br><br>
            <input type="submit" value="Send message">
          </form>
       </div>
   </body>
</html>
```

delete.js

```
var express = require('express');
var app = express();
var ejs = require('ejs');
var mysql = require('mysql');
var bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({ extended: true }));

var con = mysql.createConnection({
   host: "localhost",
```

```
   user: "root",
   password: "",
   database: "julkar"
  });
app.get('/',function(req,res,next){
   res.sendfile('views/delete.html');
});
app.post('/delete', function(req, res)
{
console.log('req.body');
console.log(req.body);
res.write('You sent the ID "' + req.body.id+'".\n');
res.end()

con.connect(function(err) {
   if (err) throw err;
   var sql = "delete from info where id='"+req.body.id+"'";
   con.query(sql, function (err, result) {
     if (err) throw err;
     console.log("1 record deleted");
      res.end();
```

```
    });
    });
});
app.listen(3000);
console.log('Example app listening at port:3000');
```

update.js

```
var express = require('express');
var app = express();
var ejs = require('ejs');
var mysql = require('mysql');
var bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({ extended: true }));

var con = mysql.createConnection({
    host: "localhost",
    user: "root",
    password: "",
    database: "julkar"
  });
```

```
app.get('/',function(req,res,next){
   res.sendfile('views/update.html');
});
app.post('/update', function(req, res)
{
console.log('req.body');
console.log(req.body);
res.write('You sent the ID "' + req.body.id+'".\n');
res.write('You sent the name "' + req.body.name+'".\n');

res.write('You sent the Email "' + req.body.email+'".\n');
res.write('You sent the City "' + req.body.city+'".\n');
res.write('You sent the Pincode "' + req.body.pincode+'".\n');
res.end()

con.connect(function(err) {
   if (err) throw err;
   var sql = "update info set name='"+req.body.name+"',email='"+
req.body.email+"',city='"+req.body.city+"',pincode='"+req.body.pin
code+"' where id='"+req.body.id+"'";
   con.query(sql, function (err, result) {
     if (err) throw err;
```

```
      console.log("1 record updated");
       res.end();
    });
    });
});
app.listen(3000);
console.log('Example app listening at port:3000');
```