

Node.js is a server-side platform built on Google Chrome's JavaScript Engine (V8 Engine). Node.js was developed by Ryan Dahl in 2009 and its latest version is v0.10.36. The definition of Node.js as supplied by its [official documentation](https://nodejs.org/) is as follows −

Node.js is a platform built on [Chrome's JavaScript runtime](https://code.google.com/p/v8/) for easily building fast and scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

Node.js is an open source, cross-platform runtime environment for developing server-side and networking applications. Node.js applications are written in JavaScript, and can be run within the Node.js runtime on OS X, Microsoft Windows, and Linux.

Node.js also provides a rich library of various JavaScript modules which simplifies the development of web applications using Node.js to a great extent.

Node.js is based on an event-driven architecture and a non-blocking Input/Output API that is designed to optimize an application's throughput and scalability for real-time web applications.

Over a long period of time, the framework available for web development was all based on a stateless model. A stateless model is where the data generated in one session (such as information about user settings and events that occurred) is not maintained for usage in the next session with that user.

A lot of work had to be done to maintain the session information between requests for a user. But with Node.js there is finally a way for web applications to have a real-time, two-way connection, where both the client and server can initiate communication, allowing them to exchange data freely.

Node.js = Runtime Environment + JavaScript Library

**Why use Node.js?**

Over the years, most of the applications were based on a statelessrequest-response framework. In these sort of applications, it is up to the developer to ensure the right code was put in place to ensure the state of web session was maintained while the user was working with the system.

But with Node.js web applications, you can now work in real-time and have a 2-way communication. The state is maintained, and the either the client or server can start the communication.

**Features of Node.js:**

Following are some of the important features that make Node.js the first choice of software architects.

* **Asynchronous and Event Driven** − All APIs of Node.js library are asynchronous, that is, non-blocking. It essentially means a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call.
* **Very Fast** − Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.
* **Single Threaded but Highly Scalable** − Node.js uses a single threaded model with event looping. Event mechanism helps the server to respond in a non-blocking way and makes the server highly scalable as opposed to traditional servers which create limited threads to handle requests. Node.js uses a single threaded program and the same program can provide service to a much larger number of requests than traditional servers like Apache HTTP Server.
* **No Buffering** − Node.js applications never buffer any data. These applications simply output the data in chunks.
* **License** − Node.js is released under the [MIT license](https://raw.githubusercontent.com/joyent/node/v0.12.0/LICENSE).

**Who Uses Node.js?**

Following is the link on github wiki containing an exhaustive list of projects, application and companies which are using Node.js. This list includes eBay, General Electric, GoDaddy, Microsoft, PayPal, Uber, Wikipins, Yahoo!, and Yammer to name a few.

**Concepts :**

The following diagram depicts some important parts of Node.js which we will discuss in detail in the subsequent chapters.



**Where to Use Node.js?**

Following are the areas where Node.js is proving itself as a perfect technology partner.

* I/O bound Applications
* Data Streaming Applications
* Data Intensive Real-time Applications (DIRT)
* JSON APIs based Applications
* Single Page Applications
* Chat applications
* **Game servers** – Fast and high-performance servers that need to processes thousands of requests at a time, then this is an ideal framework.
* **Good for collaborative environment** – This is good for environments which manage document. In document management environment you will have multiple people who post their documents and do constant changes by checking out and checking in documents. So Node.js is good for these environments because the event loop in Node.js can be triggered whenever documents are changed in a document managed environment.
* **Advertisement servers** – Again here you could have thousands of request to pull advertisements from the central server and Node.js can be an ideal framework to handle this.
* **Streaming servers** – Another ideal scenario to use Node is for multimedia streaming servers wherein clients have request's to pull different multimedia contents from this server.

Node.js is good when you need high levels of concurrency but less amount of dedicated CPU time.

Best of all, since Node.js is built on javascript, it's best suited when you build client-side applications which are based on the same javascript framework.

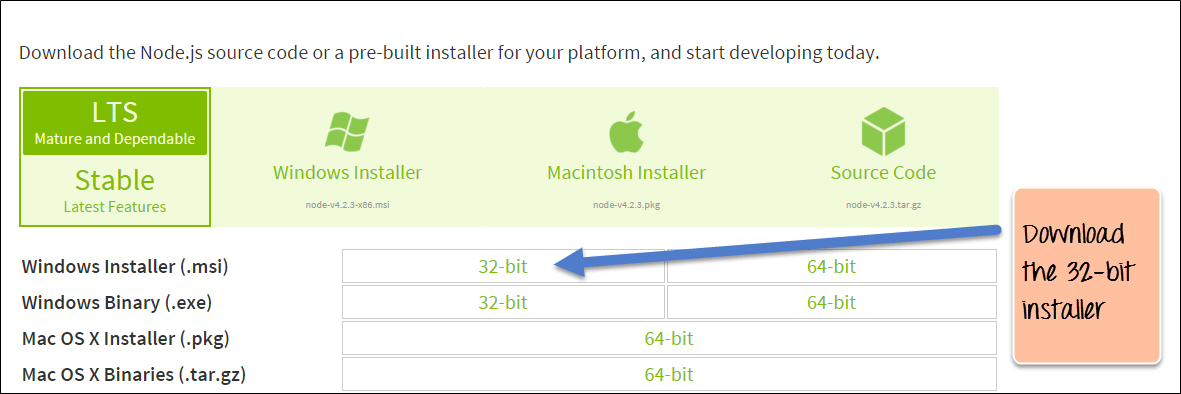
**When to not use Node.js:**

Node.js can be used for a lot of applications with various purpose, the only scenario where it should not be used is if there are long processing times which is required by the application.

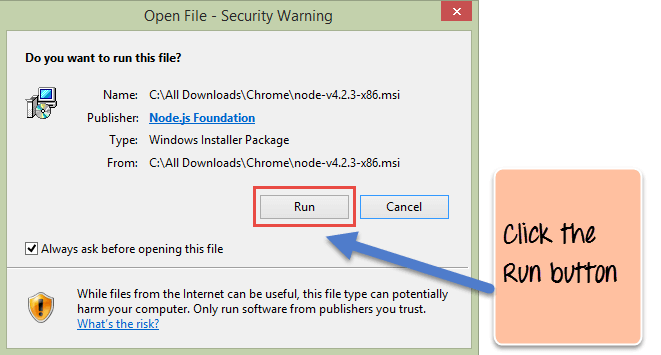
Node is structured to be single threaded. If any application is required to carry out some long running calculations in the background. So if the server is doing some calculation, it won't be able to process any other requests. As discussed above, Node.js is best when processing needs less dedicated CPU time.

# How to Download & Install Node.js - NPM on Windows

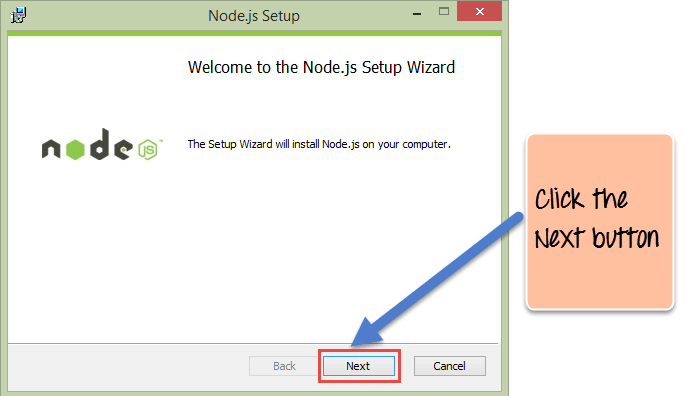
1. Go to the site <https://nodejs.org/en/download/> and download the necessary binary files. In our example, we are going to the download the 32-bit setup files for Node.js.

[](https://cdn.guru99.com/images/NodeJS/010716_0458_DownloadIns1.png)

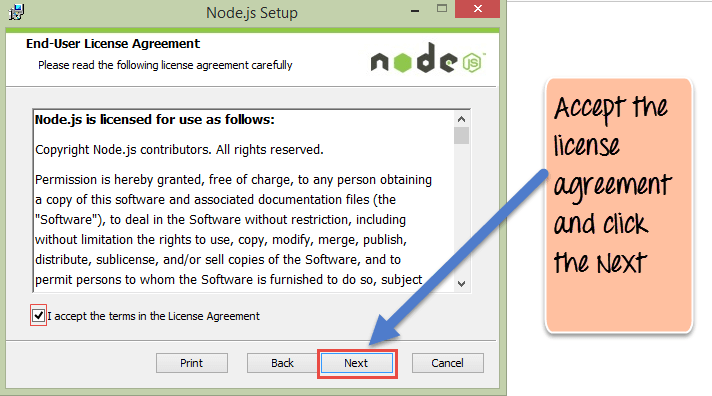
1. Double click on the downloaded .msi file to start the installation. Click the Run button in the first screen to begin the installation.

[](https://cdn.guru99.com/images/NodeJS/010716_0458_DownloadIns2.png)

1. In the next screen, click the "Next" button to continue with the installation

[](https://cdn.guru99.com/images/NodeJS/010716_0458_DownloadIns3.png)

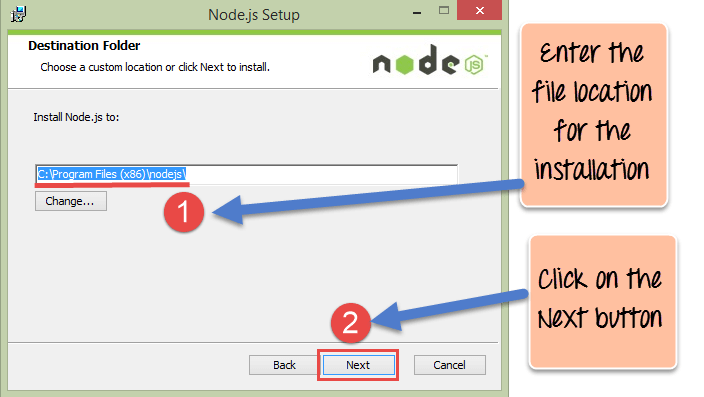
1. In the next screen Accept the license agreement and click on the Next button.

[](https://cdn.guru99.com/images/NodeJS/010716_0458_DownloadIns4.png)

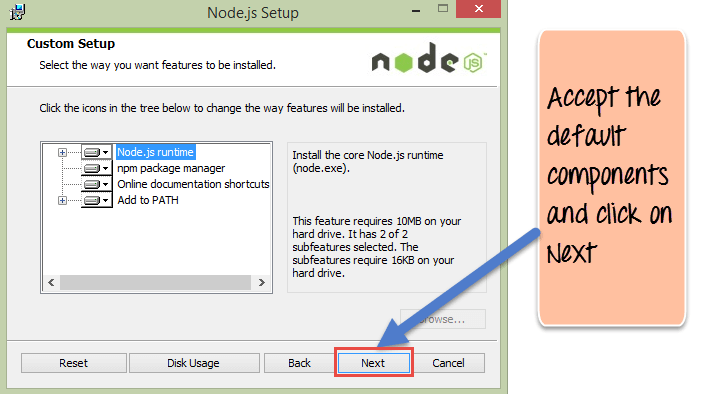
1. In the next screen, choose the location where Node.js needs to be installed and then click on the Next button.

1.First enter the file location for the installation of Node.js. This is where the files for Node.js will be stored after the installation.

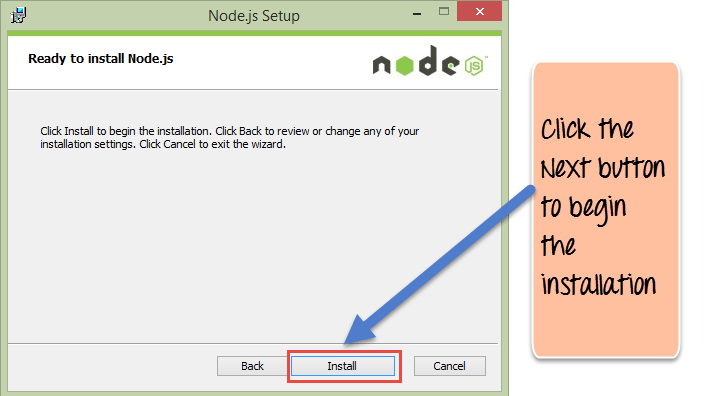
2.Click on the Next button to proceed ahead with the installation.

[](https://cdn.guru99.com/images/NodeJS/010716_0458_DownloadIns5.png)

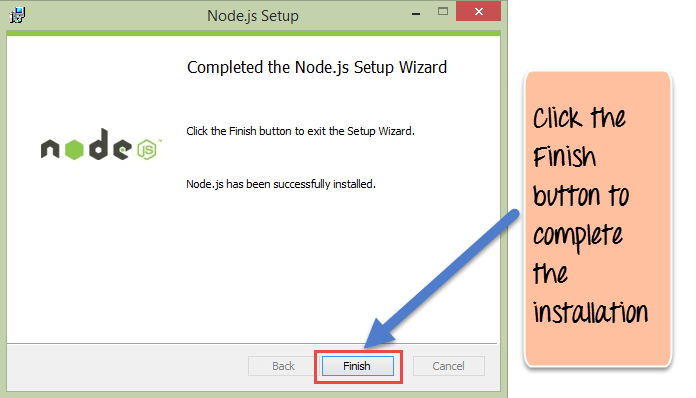
1. Accept the default components and click on the next button.

[](https://cdn.guru99.com/images/NodeJS/010716_0458_DownloadIns6.png)

1. In the next screen, click the Install button to start the installation.

[](https://cdn.guru99.com/images/NodeJS/010716_0458_DownloadIns7.png)

1. Click the Finish button to complete the installation.



# Running Hello world application in Node.js

A Node.js application consists of the following three important components

* **Import required modules** − We use the **require** directive to load Node.js modules.
* **Create server** − A server which will listen to client's requests similar to Apache HTTP Server.
* **Read request and return response** − The server created in an earlier step will read the HTTP request made by the client which can be a browser or a console and return the response.

### Step 1 - Import Required Module

We use the **require** directive to load the http module and store the returned HTTP instance into an http variable as follows −

var http = require("http");

### Step 2 - Create Server

We use the created http instance and call **http.createServer()** method to create a server instance and then we bind it at port 8081 using the **listen** method associated with the server instance. Pass it a function with parameters request and response. Write the sample implementation to always return "Hello World".

http.createServer(function (request, response) {

// Send the HTTP header

// HTTP Status: 200 : OK

// Content Type: text/plain

response.writeHead(200, {'Content-Type': 'text/plain'});

// Send the response body as "Hello World"

response.end('Hello World\n');

}).listen(8081);

console.log('Server running at http://127.0.0.1:8081/');

The above code is enough to create an HTTP server which listens, i.e., waits for a request over 8081 port on the local machine.

### Step 3 - Testing Request & Response

Let's put step 1 and 2 together in a file called **main.js** and start our HTTP server as shown below −

var http = require("http");

http.createServer(function (request, response) {

// Send the HTTP header

// HTTP Status: 200 : OK

// Content Type: text/plain

response.writeHead(200, {'Content-Type': 'text/plain'});

// Send the response body as "Hello World"

response.end('Hello World\n');

}).listen(8081);

// Console will print the message

console.log('Server running at http://127.0.0.1:8081/');

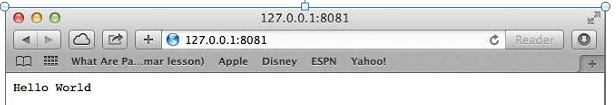
Now execute the main.js to start the server as follows −

$ node main.js

Verify the Output. Server has started.

Server running at http://127.0.0.1:8081/

## Make a Request to the Node.js Server

Open http://127.0.0.1:8081/ in any browser and observe the following result

# Create, Publish, Extend & Manage Modules:

**A] What are modules in Node.js?**

As stated earlier, modules in Node js are a way of encapsulating code in a separate logical unit. There are many readymade modules available in the market which can be used within Node js.

Below are some of the popular modules which are used in a Node js application

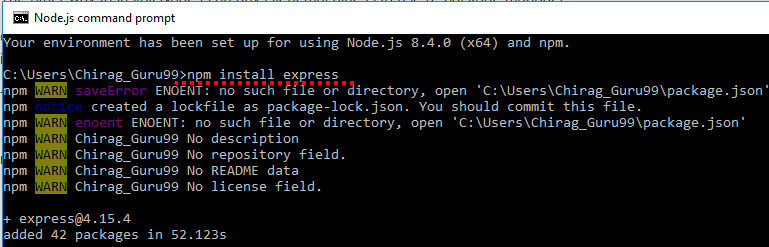
1. **Express framework** – Express is a minimal and flexible Node js web application framework that provides a robust set of features for the web and[Mobile](https://www.guru99.com/mobile-testing.html)applications.
2. **Socket.io** - Socket.IO enables real-time bidirectional event-based communication. This module is good for creation of chatting based applications.
3. **Jade** - Jade is a high-performance template engine and implemented with[JavaScript](https://www.guru99.com/interactive-javascript-tutorials.html)for node and browsers.
4. **MongoDB** - The[MongoDB](https://www.guru99.com/mongodb-tutorials.html)Node.js driver is the officially supported node.js driver for MongoDB.
5. **Restify** - Restify is a lightweight framework, similar to express for building REST APIs
6. **Bluebird** - Bluebird is a fully featured promise library with focus on innovative features and performance

**B] Using modules in Node.js**

In order to use modules in a Node.js application, they first need to be installed using the Node package manager.

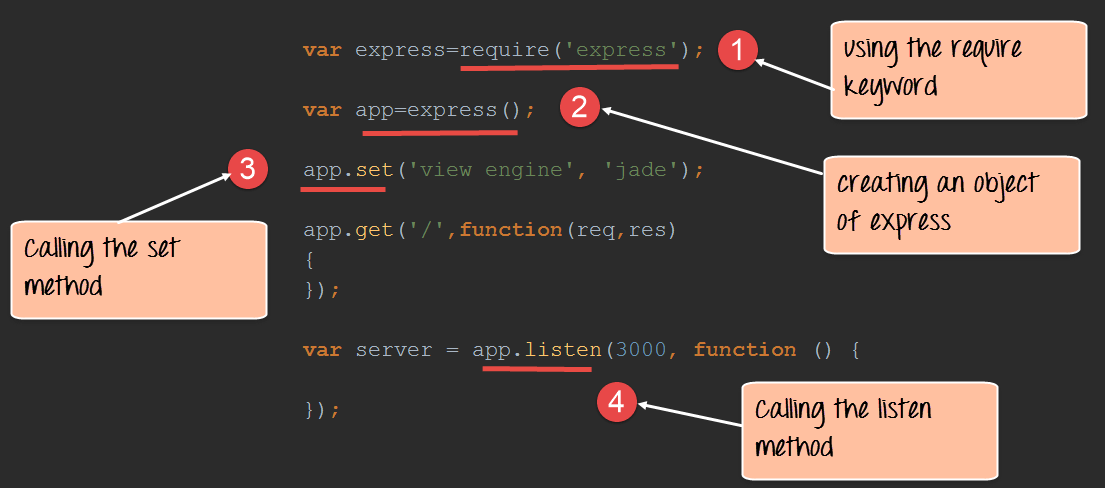
The below command line shows how a module "express" can be installed.

npm install express

[](https://cdn.guru99.com/images/NodeJS/npm_install_express.png)

* The above command will download the necessary files which contain the "express modules" and take care of the installation as well
* Once the module has been installed, in order to use a module in a Node.js application you need to use the 'require' keyword. This keyword is a way that Node.js uses to incorporate the functionality of a module in an application.

Let's look at an example how we can use the "require" keyword. The below "Guru99" code example shows how to use the require function

[](https://cdn.guru99.com/images/NodeJS/010716_0523_NodejsModul1.png)

var express=require('express');

var app=express();

app.set('view emngine','jade');

app.get('/',function(req,res)

{

});

var server=app.listen(3000,function()

{

});

1. In the first statement itself, we are using the "require" keyword to include the express module. The "express" module is an optimized JavaScript library for Node.js development. This is one of the most commonly used Node.js modules.
2. After the module is included, in order to use the functionality within the module, an object needs to be created. Here an object of the express module is created.
3. Once the module is included using the "require" command and an "object" is created, the required methods of the express module can be invoked. Here we are using the set command to set the view engine, which is used to set the templating engine used in Node.js.

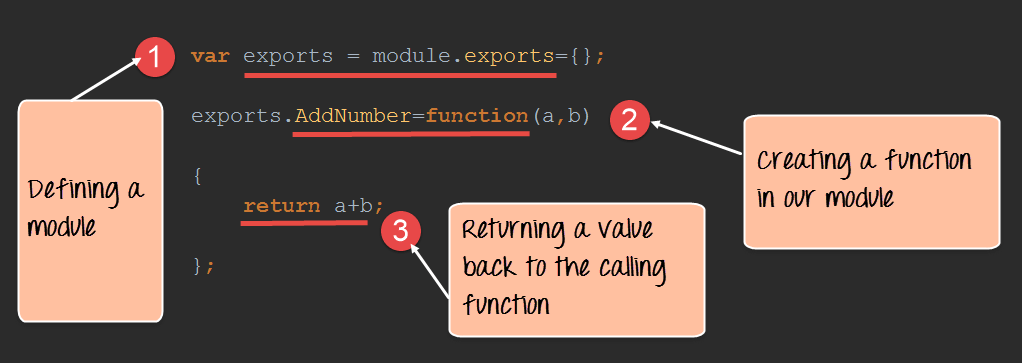
**C] Creating NPM modules**

Node.js has the ability to create custom modules and allows you to include those custom modules in your Node.js application.Let's look at a simple example of how we can create our own module and include that module in our main application file. Our module will just do a simple task of adding 2 numbers.

Let's follow the below steps to see how we can create modules and include them in our application.

Step 1) Create a file called "Addition.js" and include the below code. This file will contain the logic for your module.

Below is the code which would go into this file;

[](https://cdn.guru99.com/images/NodeJS/010716_0523_NodejsModul2.png)

var exports=module.exports={};

exports.AddNumber=function(a,b)

{

return a+b;

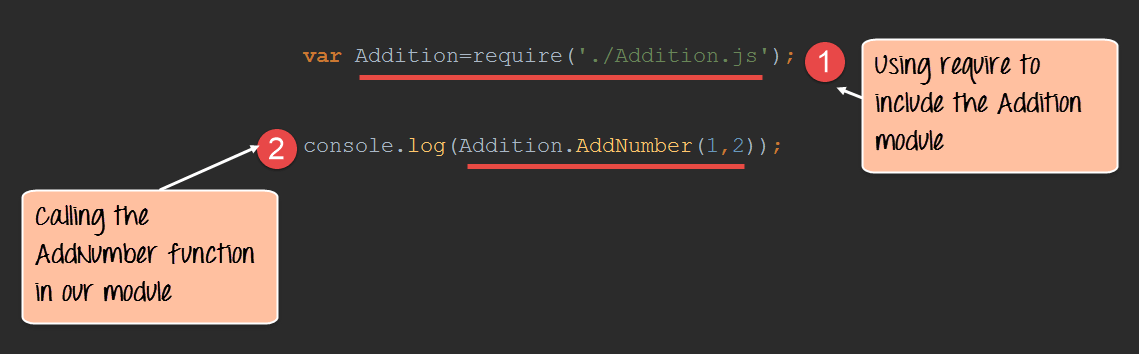
};

1. The "exports" keyword is used to ensure that the functionality defined in this file can actually be accessed by other files.
2. We are then defining a function called 'AddNumber'. This function is defined to take 2 parameters, a and b. The function is added to the module "exports" to make the function as a public function that can be accessed by other application modules.
3. We are finally making our function return the added value of the parameters.

Now that we have created our custom module which has the functionality of adding 2 numbers. It's now time to create an application, which will call this module.

In the next step, we will actually see how to create the application which will call our custom module.

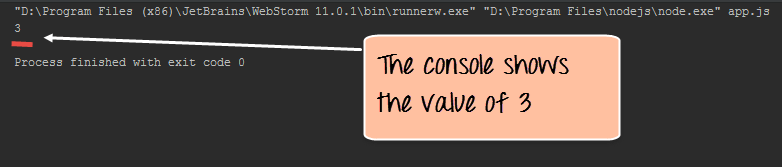
Step 2) Create a file called "app.js," which is your main application file and add the below code

[](https://cdn.guru99.com/images/NodeJS/010716_0523_NodejsModul3.png)

var Addition=require('./Addition.js');

console.log(Addition.AddNumber(1,2));

1. We are using the "require" keyword to include the functionality in the Addition.js file.
2. Since the functions in the Addition.js file are now accessible, we can now make a call to the AddNumber function. In the function, we are passing 2 numbers as parameters. We are then displaying the value in the console.

[](https://cdn.guru99.com/images/NodeJS/010716_0523_NodejsModul4.png)

**Output**:

1. When you run the app.js file, you will get an output of value 3 in the console log.
2. The result is because the AddNumber function in the Addition.js file was called successfully and the returned value of 3 was displayed in the console.

Note: - We are not using the "Node package manager" as of yet to install our Addition.js module. This is because the module is already part of our project on the local machine. The Node package manager comes in the picture when you publish a module on the internet which we see in the subsequent topic.

**D] Extending modules**

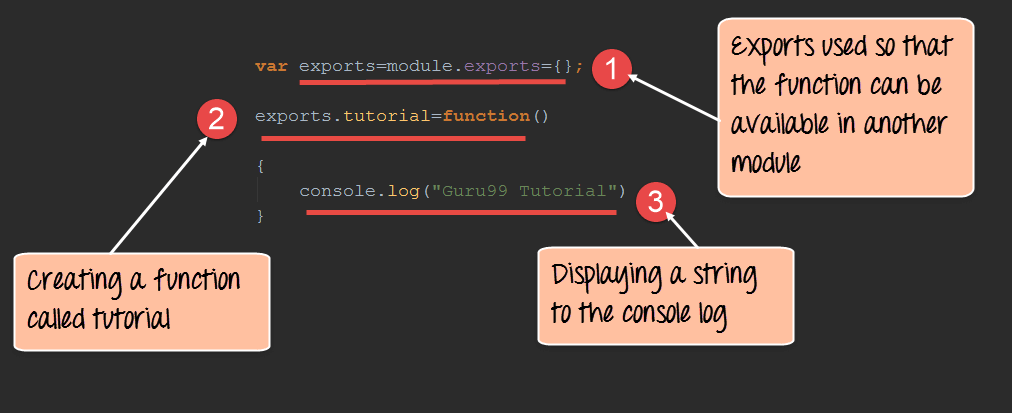
When creating modules, it is also possible to extend or inherit one module from another.In modern day programming, it's quite common to build a library of common modules and then extend the functionality of these common modules if required.

Let's look at an example of how we can extend modules in Node.js.

Step 1) Create the base module.

In our example, create a file called "Tutorial.js" and place the below code.

In this code, we are just creating a function which returns a string to the console. The string returned is "Guru99 Tutorial".

[[](https://cdn.guru99.com/images/NodeJS/010716_0523_NodejsModul5.png)](https://cdn.guru99.com/images/NodeJS/010716_0523_NodejsModul5.png)

var exports=module.exports={};

exports.tutorial=function()

{

console.log("Guru99 Tutotial")

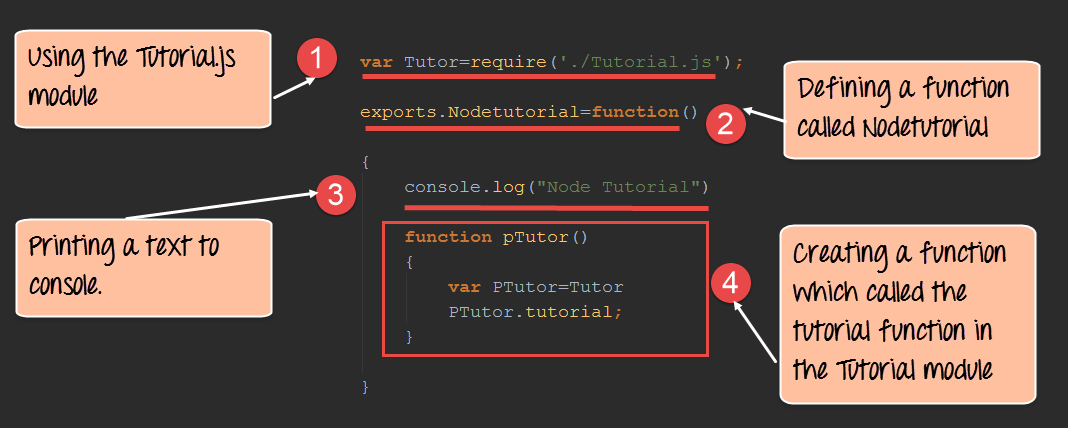
}

1. The exports module is used so that whatever function is defined in this file can be available in other modules in Node.js
2. We are creating a function called tutorial which can be used in other Node.js modules.
3. We are displaying a string "Guru99 Tutorial" in the console when this function is called.

Now that we have created our base module called Tutorial.js. It's now time to create another module which will extend this base module.

We will explore how to do this in the next step.

Step 2) Next we will create our extended module. Create a new file called "NodeTutorial.js" and place the below code in the file.

[](https://cdn.guru99.com/images/NodeJS/010716_0523_NodejsModul6.png)

var Tutor=require('./Tutorial.js');

exports.NodeTutorial=function()

{

console.log("Node Tutorial")

function pTutor()

{

var PTutor=Tutor

PTutor.tutorial;

}

}

Note, the following key points about the above code

We are using the "require" function in the new module file itself. Since we are going to extend the existing module file "Tutorial.js", we need to first include it before extending it.

We then create a function called "Nodetutorial." This function will do 2 things,

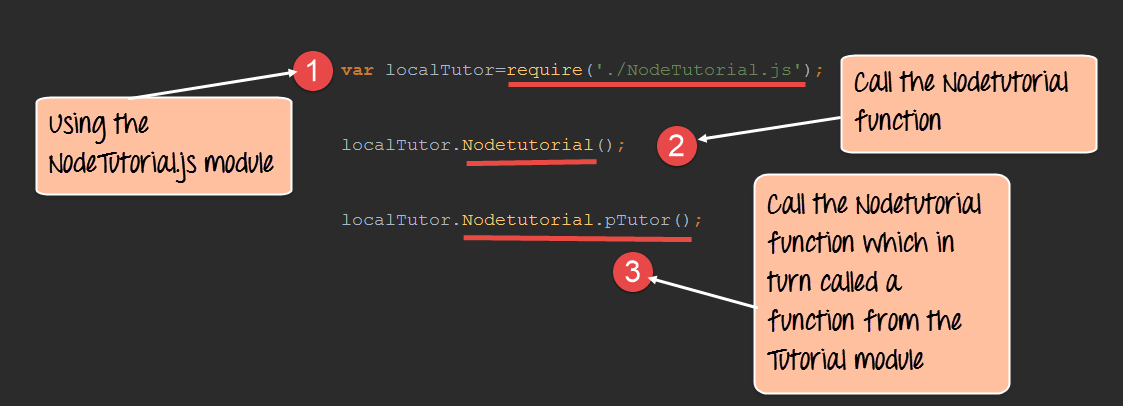
It will send a string "Node Tutorial" to the console.

It will send the string "Guru99 Tutorial" from the base module "Tutorial.js" to our extended module "NodeTutorial.js".

Here we are carrying out the first step to send a string to "Node Tutorial" to the console.

The next step is to call the function from our Tutorial module, which will output the string "Guru99 Tutorial" to the console.log.

Step 3) Create your main app.js file which is your main application file and include the below code.

[](https://cdn.guru99.com/images/NodeJS/010716_0523_NodejsModul7.png)

var localTutor=require('./NodeTutorial.js');

localTutor.NodeTutorial();

localTutor.NodeTutorial.pTutor();

The above code does the following things;

Our main application file now calls the "NodeTutorial" module.

We are calling the "NodeTutorial" function. By calling this function, the text "Node Tutorial" will be displayed in the console log.

Since we have extended our Tutorial.js module and exposed a function called pTutor. It also calls the tutorial module in the Tutorial.js module, and the text "Guru99 Tutorial" will be displayed to the console as well.

**Output:**

Since we have executed the above app.js code using Node, we will get the following output in the console.log file

* Node Tutorial
* Guru99 Tutorial

**E] Publishing NPM (Node Package Manager) Modules**

One can publish their own module to their own Github repository.

By publishing your module to a central location, you are then not burdened with having to install yourself on every machine that requires it.

Instead, you can use the install command of npm and install your published npm module.

The following steps need to be followed to publish your npm module

Step 1) Create your repository on GitHub (an online code repository management tool). It can be used for hosting your code repositories.

Step 2) You need to tell your local npm installation on who you are. Which means that we need to tell npm who is the author of this module, what is the email id and any company URL, which is available which needs to be associated with this id. All of these details will be added to your npm module when it is published.

The below commands sets the name, email and URL of the author of the npm module.

npm set init.author.name "Guru99."

npm set init.author.email "[guru99@gmail.com](mailto:guru99@gmail.com)"

npm set init.author.url [http://Guru99.com](http://guru99.com/)

Step 3) The next step is to login into npm using the credentials provided in the last step. To login, you need to use the below command

npm login

Step 4) Initialize your package – The next step is to initialize the package to create the package.json file. This can be done by issuing the below command

npm init

When you issue the above command, you will be prompted for some questions. The most important one is the version number for your module.

**Step 5)** Publish to GitHub – The next step is to publish your source files to GitHub. This can be done by running the below commands.

git add.

git commit -m "Initial release"

git tag v0.0.1

git push origin master --tags

**Step 6)** Publish your module – The final bit is to publish your module into the npm registry. This is done via the below command.

npm publish

## F] Managing third party packages with npm

As we have seen, the "Node package manager" has the ability to manage modules, which are required by Node.js applications.

Let's look at some of the functions available in the node package manager for managing modules

1. Installing packages in global mode – Modules can be installed at the global level, which just basically means that these modules would be available for all Node.js projects on a local machine. The example below shows how to install the "express module" with the global option.

npm install express –global

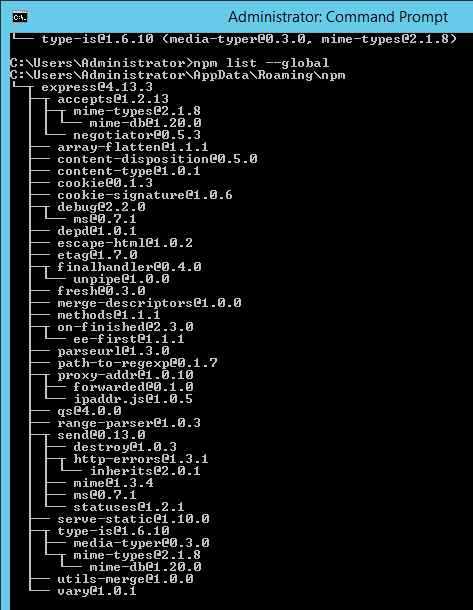
The global option in the above statement is what allows the modules to be installed at a global level.

1. Listing all of the global packages installed on a local machine. This can be done by executing the below command in the command prompt

npm list --global

Below is the output which will be shown, if you have previously installed the "express module" on your system.

Here you can see the different modules installed on the local machine.

[](https://cdn.guru99.com/images/NodeJS/010716_0523_NodejsModul8.png)

1. Installing a specific version of a package – Sometimes there may be a requirement to install just the specific version of a package. Once you know what is the package and the relevant version that needs to be installed, you can use the npm install command to install that specific version.
2. The example below shows how to install the module called underscore with a specific version of 1.7.0
3. npm install underscore@1.7.0
4. Updating a package version – Sometimes you may have an older version of a package in a system, and you may want to update to the latest one available in the market. To do this one can use the npm update command. The example below shows how to update the underscore package to the latest version
5. npm update underscore
6. Searching for a particular package – To search whether a particular version is available on the local system or not, you can use the search command of npm. The example below will check if the express module is installed on the local machine or not.
7. npm search express
8. Un-installing a package – The same in which you can install a package, you can also un-install a package. The uninstallation of a package is done with the uninstallation command of npm. The example below shows how to uninstall the express module
9. npm uninstall express

## G] What is the package.json file

The "package.json" file is used to hold the metadata about a particular project. This information provides the Node package manager the necessary information to understand how the project should be handled along with its dependencies.

The package.json files contains information such as the project description, the version of the project in a particular distribution, license information, and configuration data.

The package.json file is normally located at the root directory of a Node.js project.

package.json is present in the root directory of any Node application/module and is used to define the properties of a package. Let's open package.json of express package present in **node\_modules/express/**

## Attributes of Package.json

* **name** − name of the package
* **version** − version of the package
* **description** − description of the package
* **homepage** − homepage of the package
* **author** − author of the package
* **contributors** − name of the contributors to the package
* **dependencies** − list of dependencies. NPM automatically installs all the dependencies mentioned here in the node\_module folder of the package.
* **repository** − repository type and URL of the package
* **main** − entry point of the package
* **keywords** − keywords

# Node.js - RESTful API

# 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.

## 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.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S. N.** | **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.

## 1) 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

}

}

## 2) 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.post('/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}

}

## 3) 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}

## 4) 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.delete('/deleteUser', 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}}

# Node.js - Web Module

**What is a Web Server?**

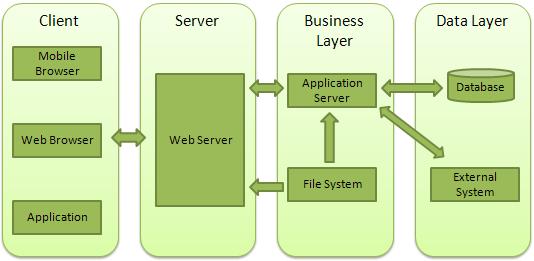
A Web Server is a software application which handles HTTP requests sent by the HTTP client, like web browsers, and returns web pages in response to the clients. Web servers usually deliver html documents along with images, style sheets, and scripts.

Most of the web servers support server-side scripts, using scripting languages or redirecting the task to an application server which retrieves data from a database and performs complex logic and then sends a result to the HTTP client through the Web server.

Apache web server is one of the most commonly used web servers. It is an open source project.

**Web Application Architecture**

A Web application is usually divided into four layers −



* **Client** − This layer consists of web browsers, mobile browsers or applications which can make HTTP requests to the web server.
* **Server** − This layer has the Web server which can intercept the requests made by the clients and pass them the response.
* **Business** − This layer contains the application server which is utilized by the web server to do the required processing. This layer interacts with the data layer via the database or some external programs.
* **Data** − This layer contains the databases or any other source of data.

**A] Creating a Web Server using Node**

Node.js provides an **http** module which can be used to create an HTTP client of a server. Following is the bare minimum structure of the HTTP server which listens at 8081 port.

Create a js file named server.js −

**File: server.js**

var http = require('http');

var fs = require('fs');

var url = require('url');

// Create a server

http.createServer( function (request, response) {

// Parse the request containing file name

var pathname = url.parse(request.url).pathname;

// Print the name of the file for which request is made.

console.log("Request for " + pathname + " received.");

// Read the requested file content from file system

fs.readFile(pathname.substr(1), function (err, data) {

if (err) {

console.log(err);

// HTTP Status: 404 : NOT FOUND

// Content Type: text/plain

response.writeHead(404, {'Content-Type': 'text/html'});

}else {

//Page found

// HTTP Status: 200 : OK

// Content Type: text/plain

response.writeHead(200, {'Content-Type': 'text/html'});

// Write the content of the file to response body

response.write(data.toString());

}

// Send the response body

response.end();

});

}).listen(8081);

// Console will print the message

console.log('Server running at http://127.0.0.1:8081/');

Next let's create the following html file named index.htm in the same directory where you created server.js.

**File: index.htm**

<html>

<head>

<title>Sample Page</title>

</head>

<body>

Hello World!

</body>

</html>

Now let us run the server.js to see the result −

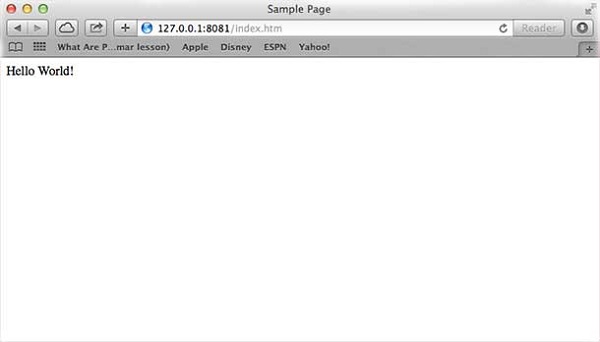
$ node server.js

Verify the Output.

Server running at http://127.0.0.1:8081/

**Make a request to Node.js server**

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



Verify the Output at server end.

Server running at http://127.0.0.1:8081/

Request for /index.htm received.

**B] Creating Web client using Node**

A web client can be created using **http** module. Let's check the following example.

Create a js file named client.js −

**File: client.js**

var http = require('http');

// Options to be used by request

var options = {

host: 'localhost',

port: '8081',

path: '/index.htm'

};

// Callback function is used to deal with response

var callback = function(response){

// Continuously update stream with data

var body = '';

response.on('data', function(data) {

body += data;

});

response.on('end', function() {

// Data received completely.

console.log(body);

});

}

// Make a request to the server

var req = http.request(options, callback);

req.end();

Now run the client.js from a different command terminal other than server.js to see the result −

$ node client.js

Verify the Output.

<html>

<head>

<title>Sample Page</title>

</head>

<body>

Hello World!

</body>

</html>

Verify the Output at server end.

Server running at http://127.0.0.1:8081/

Request for /index.htm received.

# Node.js - Mysql

* 1. **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

* 1. **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 options = {

host: "localhost",

user: "root",

password: ""

};  
  
var con = mysql.createConnection(options);  
  
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");  
});

* 1. **Creating a Database**

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

var mysql = require('mysql');  
var options = {

host: "localhost",

user: "root",

password: ""

};  
  
var con = mysql.createConnection(options);  
  
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");  
  con.query("**CREATE DATABASE nodejs**", function (err, result) {  
    if (err) throw err;  
    console.log("Database created");  
  });  
});

* 1. **Create Table**

var mysql = require('mysql');  
var options = {

host: "localhost",

user: "root",

password: "",

database : “nodejs”

};  
  
var con = mysql.createConnection(options);  
  
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");  
  });  
});

* 1. **Insert Into Table**

var mysql = require('mysql');  
var options = {

host: "localhost",

user: "root",

password: "",

database : “nodejs”

};  
  
var con = mysql.createConnection(options);  
  
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("Data Inserted");  
  });  
});

* 1. **Select From**

var mysql = require('mysql');  
var options = {

host: "localhost",

user: "root",

password: "",

database : “nodejs”

};  
  
var con = mysql.createConnection(options);  
  
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");

**var sql = "select \* from customers";**

con.query(sql, function (err, result, fields) {  
    if (err) throw err;  
    console.log(result);

    console.log(fields); //This will print the  information about each field in the result  
  
  });  
});

* 1. **Where**

var mysql = require('mysql');  
var options = {

host: "localhost",

user: "root",

password: "",

database : “nodejs”

};  
  
var con = mysql.createConnection(options);  
  
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");

**var sql = "select \* from customers where address = 'Highway 37' ";**

con.query(sql, function (err, result, fields) {  
    if (err) throw err;  
    console.log(result);

  });  
});

* 1. **Order by**
* Use the ORDER BY statement to sort the result in ascending or descending order.
* The ORDER BY keyword sorts the result ascending by default. To sort the result in descending order, use the DESC keyword.

var mysql = require('mysql');  
var options = {

host: "localhost",

user: "root",

password: "",

database : “nodejs”

};  
  
var con = mysql.createConnection(options);  
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");

**var sql = "select \* from customers order by name ";**

con.query(sql, function (err, result, fields) {  
    if (err) throw err;  
    console.log(result);

  });  
});

**9) Delete**

var mysql = require('mysql');  
var options = {

host: "localhost",

user: "root",

password: "",

database : “nodejs”

};  
  
var con = mysql.createConnection(options);  
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");

**var sql = "delete from customers where address='Pune' ";**

con.query(sql, function (err, result, fields) {  
    if (err) throw err;  
    console.log(result);

  });  
});

**10) Update**

var mysql = require('mysql');  
var options = {

host: "localhost",

user: "root",

password: "",

database : “nodejs”

};  
  
var con = mysql.createConnection(options);  
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");

**var sql = "UPDATE customers SET address = 'Bangalore' WHERE name='Wipro' ";**

con.query(sql, function (err, result, fields) {  
    if (err) throw err;  
    console.log(result);

  });  
});

**11) Drop Table**

var mysql = require('mysql');  
var options = {

host: "localhost",

user: "root",

password: "",

database : “nodejs”

};  
  
var con = mysql.createConnection(options);  
  
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");

var sql= "DROP TABLE customers";  
  con.query(sql, function (err, result) {  
    if (err) throw err;  
    console.log("Table created");  
  });  
});

**12) Drop Database**

var mysql = require('mysql');  
var options = {

host: "localhost",

user: "root",

password: "",

};  
  
var con = mysql.createConnection(options);  
  
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");

var sql= "DROP DATABASE nodejs";  
  con.query(sql, function (err, result) {  
    if (err) throw err;  
    console.log("Table created");  
  });  
});