https://nodejs.dev/learn/introduction-to-nodejs

Introduction to Node.js

TABLE OF CONTENTS

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app runs in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers - you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags.

A Vast Number of Libraries

npm with its simple structure helped the ecosystem of Node.js proliferate, and now the npm registry hosts over 1,000,000 open source packages you can freely use.

An Example Node.js Application

The most common example Hello World of Node.js is a web server:

This code first includes the Node.js `http` module.

Node.js has a fantastic standard library, including first-class support for networking.

The `createServer()` method of `http` creates a new HTTP server and returns it.

The server is set to listen on the specified port and host name. When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the `request` event is called, providing two objects: a request (an `http.IncomingMessage` object) and a response (an `http.ServerResponse` object).

Those 2 objects are essential to handle the HTTP call.

The first provides the request details. In this simple example, this is not used, but you could access the request headers and request data.

The second is used to return data to the caller.

In this case with:

```
res.statusCode = 200
```

we set the statusCode property to 200, to indicate a successful response.

We set the Content-Type header:

```
res.setHeader('Content-Type', 'text/plain')
```

and we close the response, adding the content as an argument to `end()`:

```
res.end('Hello World\n')
```

Node.js Frameworks and Tools

Node.js is a low-level platform. In order to make things easy and exciting for developers, thousands of libraries were built upon Node.js by the community.

Many of those established over time as popular options. Here is a non-comprehensive list of the ones worth learning:

- **AdonisJS**: A TypeScript-based fully featured framework highly focused on developer ergonomics, stability, and confidence. Adonis is one of the fastest Node.js web frameworks.
- **Egg.js**: A framework to build better enterprise frameworks and apps with Node.js & Koa.
- **Express**: It provides one of the most simple yet powerful ways to create a web server. Its minimalist approach, unopinionated, focused on the core features of a server, is key to its success.
- **Fastify**: A web framework highly focused on providing the best developer experience with the least overhead and a powerful plugin architecture. Fastify is one of the fastest Node.js web frameworks.
- **FeatherJS**: Feathers is a lightweight web-framework for creating real-time applications and REST APIs using JavaScript or TypeScript. Build prototypes in minutes and production-ready apps in days.
- **Gatsby**: A React-based, GraphQL powered, static site generator with a very rich ecosystem of plugins and starters.
- **hapi**: A rich framework for building applications and services that enables developers to focus on writing reusable application logic instead of spending time building infrastructure.
- **koa**: It is built by the same team behind Express, aims to be even simpler and smaller, building on top of years of knowledge. The new project born out of the need to create incompatible changes without disrupting the existing community.
- **Loopback.io**: Makes it easy to build modern applications that require complex integrations.
- **Meteor**: An incredibly powerful full-stack framework, powering you with an isomorphic approach to building apps with JavaScript, sharing code on the client and the server. Once an off-the-shelf tool that provided everything, now integrates with frontend libs React, Vue, and Angular. Can be used to create mobile apps as well.
- **Micro**: It provides a very lightweight server to create asynchronous HTTP microservices.
- **NestJS**: A TypeScript based progressive Node.js framework for building enterprise-grade efficient, reliable and scalable server-side applications.
- **Next.js**: React framework that gives you the best developer experience with all the features you need for production: hybrid static & server rendering, TypeScript support, smart bundling, route pre-fetching, and more.
- **Nx**: A toolkit for full-stack monorepo development using NestJS, Express, React, Angular, and more! Nx helps scale your development from one team building one application to many teams collaborating on multiple applications!
- **Sapper**: Sapper is a framework for building web applications of all sizes, with a beautiful development experience and flexible filesystem-based routing. Offers SSR and more!
- **Socket.io**: A real-time communication engine to build network applications.
- **Strapi**: Strapi is a flexible, open-source Headless CMS that gives developers the freedom to choose their favorite tools and frameworks while also allowing editors to easily manage and distribute their content. By making the admin panel and API extensible through a plugin system, Strapi enables the world's largest companies to accelerate content delivery while building beautiful digital experiences.

https://nodejs.dev/learn/differences-between-nodejs-and-the-browser

Differences between Node.js and the Browser

Both the browser and Node.js use JavaScript as their programming language.

Building apps that run in the browser is a completely different thing than building a Node.js application.

Despite the fact that it's always JavaScript, there are some key differences that make the experience radically different.

From the perspective of a frontend developer who extensively uses JavaScript, Node.js apps bring with them a huge advantage: the comfort of programming everything - the frontend and the backend - in a single language.

You have a huge opportunity because we know how hard it is to fully, deeply learn a programming language, and by using the same language to perform all your work on the web - both on the client and on the server, you're in a unique position of advantage.

What changes is the ecosystem.

In the browser, most of the time what you are doing is interacting with the DOM, or other Web Platform APIs like Cookies. Those do not exist in Node.js, of course. You don't have the `document`, `window` and all the other objects that are provided by the browser.

And in the browser, we don't have all the nice APIs that Node.js provides through its modules, like the filesystem access functionality.

Another big difference is that in Node.js you control the environment. Unless you are building an open source application that anyone can deploy anywhere, you know which version of Node.js you will run the application on. Compared to the browser environment, where you don't get the luxury to choose what browser your visitors will use, this is very convenient.

This means that you can write all the modern ES6-7-8-9 JavaScript that your Node.js version supports.

Since JavaScript moves so fast, but browsers can be a bit slow to upgrade, sometimes on the web you are stuck with using older JavaScript / ECMAScript releases.

You can use Babel to transform your code to be ES5-compatible before shipping it to the browser, but in Node.js, you won't need that.

Another difference is that Node.js uses the CommonJS module system, while in the browser we are starting to see the ES Modules standard being implemented.

In practice, this means that for the time being you use `require()` in Node.js and `import` in the browser.

https://nodejs.dev/learn/run-nodejs-scripts-from-the-command-line

Run Node.js scripts from the command line

The usual way to run a Node.js program is to run the `node` globally available command (once you install Node.js) and pass the name of the file you want to execute.

If your main Node.js application file is `app.js`, you can call it by typing:

```
node app.js
```

While running the command, make sure you are in the same directory which contains the `app.js` file.

https://nodejs.dev/learn/how-to-exit-from-a-nodejs-program

How to exit from a Node.js program

There are various ways to terminate a Node.js application.

When running a program in the console you can close it with `ctrl-C`, but what we want to discuss here is programmatically exiting.

Let's start with the most drastic one, and see why you're better off *not* using it.

The `process` core module provides a handy method that allows you to programmatically exit from a Node.js program: `process.exit()`.

When Node.js runs this line, the process is immediately forced to terminate.

This means that any callback that's pending, any network request still being sent, any filesystem access, or processes writing to `stdout` or `stderr` - all is going to be ungracefully terminated right away.

If this is fine for you, you can pass an integer that signals the operating system the exit code:

```
process.exit(1)
```

By default the exit code is `0`, which means success. Different exit codes have different meaning, which you might want to use in your own system to have the program communicate to other programs.

You can read more on exit codes at https://nodejs.org/api/process.html#process_exit_codes

You can also set the `process.exitCode` property:

```
process.exitCode = 1
```

and when the program ends, Node.js will return that exit code.

A program will gracefully exit when all the processing is done.

Many times with Node.js we start servers, like this HTTP server:

```
const express = require('express')
const app = express()

app.get('/', (req, res) => {
  res.send('Hi!')
})

app.listen(3000, () => console.log('Server ready'))
```

Express is a framework that uses the http module under the hood, app.listen() returns an instance of http. You would use https.createServer if you needed to serve your app using HTTPS, as app.listen only uses the http module.

This program is never going to end. If you call `process.exit()`, any currently pending or running request is going to be aborted. This is *not nice*.

In this case you need to send the command a SIGTERM signal, and handle that with the process signal handler:

Note: `process` does not require a "require", it's automatically available.

```
const express = require('express')

const app = express()

app.get('/', (req, res) => {
  res.send('Hi!')
})

const server = app.listen(3000, () => console.log('Server ready'))

process.on('SIGTERM', () => {
  server.close(() => {
    console.log('Process terminated')
  })
})
```

What are signals? Signals are a POSIX intercommunication system: a notification sent to a process in order to notify it of an event that occurred.

`SIGKILL` is the signal that tells a process to immediately terminate, and would ideally act like `process.exit()`.

`SIGTERM` is the signal that tells a process to gracefully terminate. It is the signal that's sent from process managers like `upstart` or `supervisord` and many others.

You can send this signal from inside the program, in another function:

```
process.kill(process.pid, 'SIGTERM')
```

Or from another Node.js running program, or any other app running in your system that knows the PID of the process you want to terminate.

https://nodejs.dev/learn/build-an-http-server

Build an HTTP Server

Here is a sample Hello World HTTP web server:

Let's analyze it briefly. We include the `http` module.

We use the module to create an HTTP server.

The server is set to listen on the specified port, `3000`. When the server is ready, the `listen` callback function is called.

The callback function we pass is the one that's going to be executed upon every request that comes in. Whenever a new request is received, the `request` event is called, providing two objects: a request (an `http.IncomingMessage` object) and a response (an `http.ServerResponse` object).

`request` provides the request details. Through it, we access the request headers and request data.

`response` is used to populate the data we're going to return to the client.

In this case with

```
res.statusCode = 200
```

we set the statusCode property to 200, to indicate a successful response.

We also set the Content-Type header:

```
res.setHeader('Content-Type', 'text/html')
```

and we end close the response, adding the content as an argument to `end()`:

```
res.end('<h1>Hello, World!</h1>')
```

https://nodejs.dev/learn/reading-files-with-nodejs

Reading files with Node.js

The simplest way to read a file in Node.js is to use the `fs.readFile()` method, passing it the file path, encoding and a callback function that will be called with the file data (and the error):

```
const fs = require('fs')

fs.readFile('/Users/joe/test.txt', 'utf8' , (err, data) => {
  if (err) {
    console.error(err)
    return
  }
  console.log(data)
})
```

Alternatively, you can use the synchronous version `fs.readFileSync()`:

```
const fs = require('fs')

try {
  const data = fs.readFileSync('/Users/joe/test.txt', 'utf8')
  console.log(data)
} catch (err) {
  console.error(err)
}
```

Both `fs.readFile()` and `fs.readFileSync()` read the full content of the file in memory before returning the data.

This means that big files are going to have a major impact on your memory consumption and speed of execution of the program.

In this case, a better option is to read the file content using streams.

https://www.w3schools.com/nodejs/nodejs_mysql.asp

Node.js MySQL

❮ PreviousNext ❯

Node.js can be used in database applications.

One of the most popular databases is MySQL.

MySQL Database

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

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

Install MySQL Driver

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

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

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

```
C:\Users\Your Name>npm install mysql
```

Now you have downloaded and installed a mysql database driver.

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

```
var mysql = require('mysql');
```

Create Connection

Start by creating a connection to the database.

Use the username and password from your MySQL database.

demo_db_connection.js

```
var mysql = require('mysql');

var con = mysql.createConnection({
  host: "localhost",
  user: "yourusername",
  password: "yourpassword"
});

con.connect(function(err) {
  if (err) throw err;
  console.log("Connected!");
});
```

Run example »

Save the code above in a file called "demo_db_connection.js" and run the file:

Run "demo_db_connection.js"

```
C:\Users\Your Name>node demo_db_connection.js
```

Which will give you this result:

```
Connected!
```

Now you can start querying the database using SQL statements.

Query a Database

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

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

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

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

Node.js MySQL Create Database

❮ PreviousNext ❯

Creating a Database

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

Example

Create a database named "mydb":

```
var mysql = require('mysql');

var con = mysql.createConnection({
  host: "localhost",
  user: "yourusername",
  password: "yourpassword"
});

con.connect(function(err) {
  if (err) throw err;
  console.log("Connected!");
  con.query("CREATE DATABASE mydb", function (err, result) {
```

```
    if (err) throw err;
    console.log("Database created");
  });
});
```

Run example »

Save the code above in a file called "demo_create_db.js" and run the file:

```
Run "demo_create_db.js"

C:\Users\Your Name>node demo_create_db.js
```

Which will give you this result:

```
Connected!
Database created
```

Node.js MySQL Create Table

❮ PreviousNext ❯

Creating a Table

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

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

Example

Create a table named "customers":

```
var mysql = require('mysql');

var con = mysql.createConnection({
  host: "localhost",
  user: "yourusername",
  password: "yourpassword",
```

```
  database: "mydb"
});

con.connect(function(err) {
  if (err) throw err;
  console.log("Connected!");
  var sql = "CREATE TABLE customers (name VARCHAR(255), address VARCHAR(255))";
  con.query(sql, function (err, result) {
    if (err) throw err;
    console.log("Table created");
  });
});
```

Run example »

Save the code above in a file called "demo_create_table.js" and run the file:

Run "demo_create_table.js"

```
C:\Users\Your Name>node demo_create_table.js
```

Which will give you this result:

```
Connected!
Table created
```

Primary Key

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

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

Example

Create primary key when creating the table:

```
var mysql = require('mysql');

var con = mysql.createConnection({
```

```
  host: "localhost",
  user: "yourusername",
  password: "yourpassword",
  database: "mydb"
});

con.connect(function(err) {
  if (err) throw err;
  console.log("Connected!");
  var sql = "CREATE TABLE customers (id INT AUTO_INCREMENT PRIMARY KEY, name
VARCHAR(255), address VARCHAR(255))";
  con.query(sql, function (err, result) {
    if (err) throw err;
    console.log("Table created");
  });
});
```

Run example »

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

Example

Create primary key on an existing table:

```
var mysql = require('mysql');

var con = mysql.createConnection({
  host: "localhost",
  user: "yourusername",
  password: "yourpassword",
  database: "mydb"
});

con.connect(function(err) {
  if (err) throw err;
  console.log("Connected!");
  var sql = "ALTER TABLE customers ADD COLUMN id INT AUTO_INCREMENT PRIMARY KEY";
  con.query(sql, function (err, result) {
    if (err) throw err;
    console.log("Table altered");
  });
});
```

Node.js MySQL Insert Into

❮ PreviousNext ❯

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

Run example »

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);
  });
});
```

Run example »

Save the code above in a file called "demo_db_insert_multple.js", and run the file:

Run "demo_db_insert_multiple.js"

```
C:\Users\Your Name>node demo_db_insert_multiple.js
```

Which will give you this result:

```
Connected!
Number of records inserted: 14
```

The Result Object

When executing a query, a result object is returned.

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

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

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

```
  changedRows: 0
}
```

The values of the properties can be displayed like this:

Example

Return the number of affected rows:

```
console.log(result.affectedRows)
```

Which will produce this result:

```
14
```

Get Inserted ID

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

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

Example

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

```
var mysql = require('mysql');

var con = mysql.createConnection({
  host: "localhost",
  user: "yourusername",
  password: "yourpassword",
  database: "mydb"
});

con.connect(function(err) {
  if (err) throw err;
  var sql = "INSERT INTO customers (name, address) VALUES ('Michelle', 'Blue Village 1')";
  con.query(sql, function (err, result) {
    if (err) throw err;
    console.log("1 record inserted, ID: " + result.insertId);
```

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

Run example »

Save the code above in a file called "demo_db_insert_id.js", and run the file:

Run "demo_db_insert_id.js"

```
C:\Users\Your Name>node demo_db_insert_id.js
```

Which will give you something like this in return:

```
1 record inserted, ID: 15
```

Node.js MySQL Select From

❮ PreviousNext ❯

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);
  });
});
```

Run example »

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);
  });
});
```

Run example »

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);
  });
});
```

Run example »

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: 'address',
    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
```