**JavaScript Basics #10: Node.js**

At the beginning of this course, I promised that we’ll be using JavaScript the “traditional” way, exclusively as a frontend language. But, I still like to give a brief introduction to Node.js and how to use JavaScript on the backend. The purpose of this article is to help you further understand how the server works and how it communicates with the frontend and prepares you for our future courses on Lavavel and Django.

Before we start, create a new folder on your computer. Make sure that all the installing, creating and updating that we do in this tutorial happens in this folder.

Table of Contents

* [About Node.js](https://www.ericsdevblog.com/index.php/2021/12/javascript-basics-10-node-js/#About_Nodejs)
* [Installing and Managing Packages](https://www.ericsdevblog.com/index.php/2021/12/javascript-basics-10-node-js/#Installing_and_Managing_Packages)
* [The File System Module](https://www.ericsdevblog.com/index.php/2021/12/javascript-basics-10-node-js/#The_File_System_Module)
* [The HTTP Module](https://www.ericsdevblog.com/index.php/2021/12/javascript-basics-10-node-js/#The_HTTP_Module)
* [A Simple App](https://www.ericsdevblog.com/index.php/2021/12/javascript-basics-10-node-js/#A_Simple_App)
  + [Server](https://www.ericsdevblog.com/index.php/2021/12/javascript-basics-10-node-js/#Server)
  + [Database](https://www.ericsdevblog.com/index.php/2021/12/javascript-basics-10-node-js/#Database)

**ABOUT NODE.JS**

Node.js is a JavaScript runtime that allows us to run JavaScript on almost any platform, not just the browser. After you install Node.js on your system, you will have a program named node, and you can use it to run JavaScript files like this:

node example.js

If the file example.js contains the following code:

console.log("Hello, World!");

The output will be:

"Hello, World!"

This is very similar to what we’ve seen when JavaScript is run in browsers, so I will not waste any more of your time on this.

**INSTALLING AND MANAGING PACKAGES**

When you install Node.js on your system, a package manager called npm would also be installed. Unless you are using Linux, in which case you’ll need to install it separately.

The primary use of npm is to download and manage JavaScript packages that are required by your application. A package is a piece of program that is written and published by someone else, and simply grab it and use it in your own application.

For example, you are building an app that requires the package called ini, run the following command in the terminal. Make sure you are in the correct folder!

npm install ini

When you first run this command, npm will create three different things in your working directory.

Graphical user interface, text, application

Description automatically generated

First, a new folder named “node\_modules”, which stores the package you just installed. And there are also two JSON files, package.json and package-lock.json. Both of them are used for version control. Their difference is that the package-lock.json stores the exact version of the package, while package.json stores the minimum version that is required, as well as any other information about the app. You can easily tell their difference by comparing them side by side.

Text

Description automatically generated

To use the package we just installed, use the require() method.

const {parse} = require("ini");

// We can perform some actions using the variable parse

...

If you are interested in learning more about the npm tool and how to manage packages with it, you can go to [https://npmjs.org](https://npmjs.org/) for more documentation. But for now, we don’t need to know too much about it.

**THE FILE SYSTEM MODULE**

Before we can start building our backend app, there are two JavaScript packages I’d like to introduce, the http module and the fs module. We’ll use the http module to create a server, and use the fs module to read and write to a file, which we’ll use as a database to store information.

Let’s first start with the file system (fs) module. This package is built into Node.js, so we don’t need to install anything in this case. Instead, we’ll create a new js file for the code and a txt file for the JavaScript to read and write. We’ll import the module like we talked about before.

Text

Description automatically generated with medium confidence

// import the fs module

let { readFile } = require("fs");

// specify the file we want to read as well as the charset encoding format

readFile("data.txt", "utf8", (error, text) => {

// If there is an error reading the file

if (error) throw error;

// If no error, print the content of the file

console.log(text);

});

We can also write to the file like this:

const {writeFile} = require("fs");

writeFile("data.txt", "Hello, World? Hello, World!", error => {

if (error) console.log(`${error}`);

else console.log("File written.");

});

In this case, it is not necessary to specify the encoding format. If writeFile is given a string, it will simply assume the default format, which is UTF-8.

**THE HTTP MODULE**

Another very important module we need to talk about is http, it allows us to create an HTTP server using JavaScript. For example:

const {createServer} = require("http");

let server = createServer((request, response) => {

// If a request is recieved, return a 200 OK response along with some other information

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

// The body of the HTTP response

response.write(`<h1>Hello, World!</h1>`);

// The response ends

response.end();

});

// Make the HTTP server listen on port 8000

server.listen(8000);

console.log("Listening! (port 8000)");

The variables request and response each represents an object storing the incoming and the outgoing data. For instance, you can access the url property of the request by using request.url.

This example is very simple, but in reality, the backend servers are usually more complex. So next, let’s try something more challenging. We are going to create a simple app that asks for your name, and once you submit your name, the data will be stored in a txt file, which acts as a database. When you visit the web page again, it will greet you with your name.

**A SIMPLE APP**

**Server**

Step one, we’ll create a backend without worrying about the database. Let’s create a new JavaScript file named server.js:

const { createServer } = require("http");

let server = createServer((request, response) => {

request.on('data', function(){});

request.on('end', function(){});

});

server.listen(8000);

console.log("Listening! (port 8000)");

This is very similar to our previous example, but this time we’ll use event listeners to configure the server. The first event we are listening to is data, which means when the HTTP request is transmitting data. In this case, we should extract the information we need to use from the request.

The second event is end, which means when the request is not transmitting data, in this case, the server should respond with some information.

// Initialize the variable "name"

let name = "";

request.on("data", function (chunk) {

// "chunk" is the data being transferred

name = name + chunk;

// The data is in name/value pair (name1=value1)

// So, we need to split the name and the value

name = name.split("=");

});

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

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

// For now, we'll use the data directly without a database,

// Just to test if it works

response.write(`

<h2>Hello, ${name[1]}</h2>

<p>What is your name?</p>

<form method="POST" action="example/message.html">

<p>Name: <input type="text" name="name"></p>

<p><button type="submit">Submit</button></p>

</form>

`);

response.end();

});

Run the server with the following command:

node server.js

Open our browser and go to [http://localhost:8000](http://localhost:8000/).

Graphical user interface, application

Description automatically generated

Submit your name and see if anything changes.

Graphical user interface, text, application

Description automatically generated

**Database**

However, this data is only temporary. It will be lost if you restart the server or refresh the browser. What if you want to store the data for a bit longer?

Now, we’ll create a new file called data.txt, and we’ll use it to store the name you submitted.

const { createServer } = require("http");

const fileSystem = require("fs");

let server = createServer((request, response) => {

// To make things more clear, name is used when writing to file

// myName is used when reading from file

let name = "";

let myName = "";

request.on("data", function (chunk) {

name = name + chunk;

name = name.split("=");

name = name[1];

// Write the data to data.txt

fileSystem.writeFile("data.txt", name, function (error) {

if (error) throw error;

});

});

request.on("end", function () {

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

// Read the data from file

fileSystem.readFile("data.txt", "utf8", (error, text) => {

if (error) throw error;

myName = text;

});

response.write(`

<h2>Hello, ${myName}</h2>

<p>What is your name?</p>

<form method="POST" action="example/message.html">

<p>Name: <input type="text" name="name"></p>

<p><button type="submit">Submit</button></p>

</form>

`);

response.end();

});

});

server.listen(8000);

console.log("Listening! (port 8000)");

Notice the syntax when importing the packages. const { xxx } = require('xxx') is importing a method from a package, and const xxx = require('xxx') is importing the entire package, and we can access one of the methods using xxx.methodName.

Run this server and resubmit your name, this time if you open the data.txt file, you will see that the data has been written to the file.

Graphical user interface, application

Description automatically generated