

# JavaScript Basic

### Guía #5: Web Server con NodeJS: HTTP, Express & HBS

Esta guía hace parte del curso mencionado en el título y les permitirá estudiar con más detenimiento los conceptos de clase, además de servir como instrumento de la misma. La guía contará con código base, el desarrollo conceptual de los temas y enlaces externos para el estudio independiente.

# Marco teórico

## 1. Conceptos de NodeJS

Para iniciar este módulo es necesario comprender que:

**Características de NodeJS**

1. NodeJS es Javascript orientado a desarrollo backend basado en eventos.
2. Con NodeJS podremos acceder al sistema de archivos del equipo, cosa que no se puede con JavaScript.
3. Podremos tener acceso a información del sistema operativo.
4. Podremos ejecutar, consultar y detener procesos de la máquina.

NodeJS corre soportado por el [motor v8](https://v8.dev/) de google chrome, logrando altísima eficiencia.

**Ventajas de node:**

1. Uso de sockets para comunicación *real-time* con el servidor.
2. El manejo de archivos con el FileSystem permitirá cargas simultáneas.
3. Conexiones a bases de datos y trabajo eficiente con bases de datos No-SQL.
4. Fácil creación de servicios REST.
5. NodeJS permite Nonbloking I/O.
6. Su configuración es rápida y sencilla.
7. Su gestor de paquetes (**npm**) tiene más de 470k paquetes disponibles.
8. Como está escrito en JavaScript, ahora sólo tendremos que aprender algo de sintaxis y familiarizarnos con las operaciones asíncronas y no bloqueantes.

## 2. Concepto de entradas y salidas bloqueantos y no-bloqueantes (Bloking & Nonbloking I/O)

# Desarrollo del servidor web

## 1. Inicialización del proyecto

Para arrancar vamos a crear una carpeta que de preferencia se llamará: **webServer**, la ubicación de la misma es irrelevante pues como será NodeJS quien usemos como backend, no tendremos que simular un servidor por políticas de acceso a archivos.

mkdir webServer && cd webServer

Una vez creada la carpeta y dentro de ella vamos a inicializar un proyecto de NodeJS:

npm init

Este comando nos permitirá definir cierta información para el proyecto, pero no se preocupen si posteriormente deciden cambiarla, pues podremos editar el archivo package.json.

#### package.json

Contiene toda la información de nuestro proyecto en cuanto a su configuración, la gestión de las dependencias y los scripts de automatización que el mismo requiera.

{  
 "name": "webserver",  
 "version": "1.0.0",  
 "description": "A web server demo",  
 "main": "server.js",  
 "scripts": {  
 "test": "echo \"Error: no test specified\" && exit 1"  
 },  
 "keywords": [  
 "web",  
 "server",  
 "nodejs"  
 ],  
 "author": "andres",  
 "license": "MIT"  
}

## 2. Paquete HTTP [NodeJS Http](https://nodejs.org/docs/latest-v10.x/api/http.html)

Para empezar partamos del hecho de que a este punto estamos usando el API en su versión v10.16.0, así que mucho cuidado porque en internet muchos ejemplos usan APIs anteriores.

#### server.js

//El paquete http ya vienen cargado por defecto en NodeJS  
const http = require('http');  
const port = 8888;  
  
http.createServer( (request, response)=> {  
 response.write('Este es mi super Badass web server con node');  
 response.end();  
}).listen(port);  
  
console.log(`El super badass web server está escuchando por el puerto ${port}`);

Para poner a nadar el servidor usaremos el siguiente comando:

#la extensión no hace falta pues node asume el .js  
node server

Este es un ejemplo de servidor básico con una respuesta sencilla, pero empecemos a modificar esto con un poco de sabor, pongamos unas cabeceras:

//El paquete http ya vienen cargado por defecto en NodeJS  
const http = require('http');  
const port = 8888;  
  
let user = {  
 username: "Jhon",  
 email: "jhon@doe.com",  
 password: "1234"  
}  
  
http.createServer( (request, response)=> {  
 response.writeHead(200, {'Content-Type': 'application/json'});  
 response.write(JSON.stringify(user));  
 response.end();  
}).listen(port);  
  
console.log(`El super badass web server está escuchando por el puerto ${port}`);

Http permite tener control tanto de las peticiones como de las respuestas de nuestro servidor, en los ejemplos anteriores no hicimos uso de request, pero con el más adelante podremos hacer los primeros pasos a URLs límpias.

Hasta ahora logramos una respuesta genérica para todas las peticiones al servidor y exponer recursos a peticiones específicas y mostrar también contenidos basados en vista podría ser un poco tedioso, es acá donde entra **Express.js**.

## 3. [ExpressJS](https://expressjs.com/es/)

La instalación de Express en nuestro proyecto es bastante sencilla, como el 99% de los casos con **npm**:

#i es el shorthand de install  
# --save le indica a npm que registre la dependencia en el log  
npm i express --save

Una vez instalamos la dependencia nuestro package.json se vería así:

{  
 "name": "webserver",  
 "version": "1.0.0",  
 "description": "A web server demo",  
 "main": "server.js",  
 "scripts": {  
 "test": "echo \"Error: no test specified\" && exit 1"  
 },  
 "keywords": [  
 "web",  
 "server",  
 "nodejs"  
 ],  
 "author": "andres",  
 "license": "MIT",  
 "dependencies": {  
 "express": "^4.17.1"  
 }  
}

Ahora dentro del atributo dependencies tendremos express y la versión que usamos en la aplicación.

#### Borrón y cuenta nueva

Ahora vamos a hacer lo mismo de antes, pero usando express:

const express = require('express');  
const app = express();  
const port = 8888;  
  
app.get('/', function (req, res) {  
 res.send('Este es mi super Badass web server con node');  
});  
  
app.listen(port, () => {  
 console.log(`El super badass web server está escuchando por el puerto ${port}`);  
});

Con una sintaxis mucho más amigable obtendremos el mismo resultado, pero además la petición la hemos configurado para las que llegan por el método get y al recurso /, de tal manera que si ahora entraramos a http://localhost:8888/foo recibiríamos un mensaje de error en donde el servidor no encuentra el recurso Cannot GET /foo.

### Ahora mismo usted: ¿Y si quiero una respuesta genérica por alguna razón a todo lo que venga por get?

app.get('\*', function (req, res) {  
 res.send('Este es mi super Badass web server con node');  
});

### Ahora mismo usted (de nuevo): ¿Y toca parar el proceso del servidor y volverlo a subir siempre cada vez que se actualiza?

No, para esto vamos a usar [Nodemon](https://nodemon.io), que nos permitirá monitorear los cambios en el servidor y reiniciar el servicio de manera automática.

#-g es el flag para una instalación global, por esto debemos  
#usar sudo y meter el password de cuenta administradora  
npm install -g nodemon

Una vez instalado podemos leer la documentación [acá](https://github.com/remy/nodemon#nodemon) o simplemente dar un salto de fe y creerle a:

#nodemon [your node app]  
 nodemon server

**Nota:** estoy asumiendo que seguimos en la carpeta del proyecto.

# Ahora les toca a ustedes hacer la migración del ejemplo http con respuesta JSON

Se vería algo así:

var express = require('express');  
var app = express();  
const port = 8888;  
  
let user = {  
 name: "Jhon",  
 email: "jhon@doe.com",  
 password: "1234"  
}  
  
app.get('\*', function (req, res) {  
 res.send(user); //Express se encarga de la cabecera y el parseo.  
});  
  
app.listen(port, function () {  
 console.log(`El super badass web server está escuchando por el puerto ${port}`);  
});

## 4. Servir contenidos estáticos

A menos que escribamos servicios web o APIs, la idea principal de un web server será desplegar contenidos estáticos, para ello ahora construiremos la compatibilidad con las vistas.

#### Servir un folder estático

Justo después de la creación de las constantes adicionaremos:

app.use( express.static(`${\_\_dirname}/views`) );  
app.use( express.static(`${\_\_dirname}/public`) );

Ambas carpetas no existen y debemos crearlas, además serán nuestras carpetas accesibles del servidor, ahora para servir un archivo vamos a modificar el archivo así:

const express = require('express');  
const path = require('path');  
const app = express();  
const port = 8888;  
  
app.use( express.static(`${\_\_dirname}/views`) );  
app.use( express.static(`${\_\_dirname}/public`) );  
  
app.get('/', function (req, res) {  
 console.log(\_\_dirname);  
 res.sendFile(path.join(\_\_dirname+'/views/index/index.html'));  
});  
  
app.listen(port, function () {  
 console.log(`El super badass web server está escuchando por el puerto ${port}`);  
});

Importamos path dentro del proyecto para poder tener acceso a las direcciones y usamos el método sendFile de la respuesta para responder a la petición con un archivo, en este caso ante una petición get a la ruta / responderemos con el archivo vista ubicado en res.sendFile(path.join(\_\_dirname+'/views/index/index.html'));

Ahora dentro de la carpeta public adicionaremos una carpeta css y adentro crearemos el archivo style.css.

#### index.html

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <meta http-equiv="X-UA-Compatible" content="ie=edge">  
 <title>Web Server con NodeJS | Index</title>  
 <link rel="stylesheet" href="/css/style.css">  
</head>  
<body>  
 <h1>Este es el Index del Web Server</h1>  
</body>  
</html>

#### style.css

\*{  
 margin: 0;  
 padding: 0;  
 font-family: Arial, Helvetica, sans-serif;  
 box-sizing: border-box;  
}

Ahora, notarán que la importación de la hoja de estilo tiene la ruta /css/style.css y esto será servido por express desde el recurso estático /public.



## 4. Servir contenidos dinámicos con Handlebars

Usaremos [HandlebarsJS](https://handlebarsjs.com) para poder hacer contenidos dinámicos, pero ojo, no será de manera directa pues queremos usar el motor de vistas que soporte handlebars para ExpressJS, este se conoce como [**hbs**](https://www.npmjs.com/package/hbs).

npm i hbs --save

Ahora es tiempo de configurar hbs en express, adicionando la línea:

app.set('view engine', 'html');  
app.engine('html', require('hbs').\_\_express);

después de la configuración de nuestros recursos estáticos.

### OJO, una vez configurado esto cambiará la manera en la que podremos entregar contenidos

Ahora no tendremos que complicarnos con:

app.get('/', function (req, res) {  
 console.log(\_\_dirname);  
 res.sendFile(path.join(\_\_dirname+'/views/index/index.html'));  
});

Pues en su lugar podremos ir directo al grano:

app.get('/', function (req, res) {  
 res.render('index/index');  
});

y ahora crear nuevas vistas sería tan sencillo como

mkdir views/user

y crear una vista principal para el controlador de usuarios

touch views/user/index.html

index.html

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <meta http-equiv="X-UA-Compatible" content="ie=edge">  
 <title>User</title>  
</head>  
<body>  
 <h1>User</h1>  
</body>  
</html>

Ahora si accedemos a http://localhost:8888/user/ la respuesta será:

Ahora sí a lo que vinimos, ya estamos listos para usar Handlebars.

### Uso de handlebars en la vista de Usuario

1. Cambiaremos la extensión de la vista del usuario de .html a .hbs para que el motor de render lo interprete adecuadamente.
2. Modificamos la manera en que renderizamos la vista.

app.get('/user', function (req, res) {  
 app.set('view engine', 'hbs');  
 res.render('user/index',{  
 username: "Foo"  
 });  
});

# Ejercicio

Es momento de leer un poco al respecto de las request en NodeJS para poder utilizar los parámetros de la URL como información a nuestros sitios, sobreescriba el controlador de la petición /user para que reciba como parámetro un nombre de usuario de la forma /user/:username

## 5. Parciales

Los parciales son fragmentos o componentes de nuestros sitios que usualmente se repiten en más de una vista y requieren una centralización, como es el caso de un header, un footer, hasta la misma etiqueta head. Para crear parciales con hbs simplemente requerimos registrarlos en nuestro servidor:

// Requerimos una instancia de hbs  
const hbs = require('hbs');  
  
// Express HBS engine  
hbs.registerPartials(\_\_dirname + '/views/fragments');

Por motivos de comodidad vamos a trabajar con todas las vistas en formato hbs así que cambiaremos los formatos y actualizaremos el view render por defecto a hbs:

app.set('view engine', 'hbs');  
//y como ya importamos hbs arriba removemos esta línea  
//app.engine('html', require('hbs').\_\_express);

Ahora vamos a tomar el fragmento de head del views/index/index.hbs y crearemos un fragmento views/fragments/head.hbs:

<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <meta http-equiv="X-UA-Compatible" content="ie=edge">  
 <title>{{title}}</title>  
 <link rel="stylesheet" href="vendors/bootstrap-4.3.1-dist/css/bootstrap.min.css">  
 <link rel="stylesheet" href="css/style.css">  
</head>

**Nota:** Cambiamos el título estático por una variable de handlebars.

Y ahora podremos usar el parcial en las vistas simplemente definiendo su posición así:

{{> head }}

Ahora simplemente debemos importar el parcial en cada vista y modificar server.js para pasar el título entre los parametros:

app.get('/', function (req, res) {  
 res.render('index/index',{title: "Index Controller"});  
});  
app.get('/user', function (req, res) {  
 res.render('user/index'),{title:"User Controller"};  
});  
  
app.get('/user/:user', function (req, res) {  
 app.set('view engine', 'hbs');  
   
 let u = req.params.user; console.log(u);  
   
 let user = users.find(user => user.username.toLowerCase() == u);  
 console.log(user);  
 res.render('user/detail',{title:"User Controller", user});  
});