**Variables de Entorno en Node.js**

Ahora que ya sabemos cómo utilizar MySQL con Node.js, es hora de explicar cómo retirar ciertas variables sensibles del código principal de nuestra aplicación. ¿Pero para que querríamos hacer eso?

Bueno… pongamos como ejemplo la conexión que utilizamos en el ejercicio 06 para poder acceder a MySQL desde Node.js:

**const** conexion = mysql.createConnection({

host: '127.0.0.1',

user: 'root',

password: 'qwerty1234',

port: 3306,

database: 'db\_pendientes'

});

Como puedes ver, todos los parámetros están a la vista de cualquiera que tenga acceso a nuestro código o incluso de algún atacante que logre penetrar en nuestro servidor. Esto crea una vulnerabilidad en la seguridad del sistema, ya que cualquiera que pueda ver nuestro código, sabrá cómo ingresar a la base de datos de nuestro sistema y hacer lo que quiera con ella. Ahora imagina que este sistema controlara todo el tráfico monetario de un banco… en dado caso, estaríamos en un grande problema. Las variables de entorno crean una protección extra ante esta situación.

Ahora imaginemos que tenemos nuestro código en un controlador de versiones cómo Git (el cual recomendamos encarecidamente que aprendas) y que también tenemos un equipo de desarrollo de varios integrantes y que cada uno de éstos, localmente tiene diferente usuario y diferente contraseña para acceder a su MySQL local en su computadora personal.

Si nuestra contraseña de nuestro ambiente de testing local para MySQL es qwerty1234, pero la de otro integrante de nuestro equipo es micontraseñita, esto causaría que si nosotros subimos el código al servidor en dónde esté alojado nuestro proyecto, sobrescribiríamos la contraseña de nuestro compañero y por lo tanto, cuando el obtenga los cambios, tratará de conectarse a MySQL y este le marcará error. Él se daría cuenta de que las credenciales están mal, las cambiaría, subiría los cambios al servidor que aloja el proyecto y cuando nosotros obtengamos esos cambios, ahora a nosotros nos fallaría la conexión a nuestro MySQL local. Las variables de entorno también nos proporcionan una solución a esta situación, ya que cada uno tendrá un archivo con sus variables de entorno que nunca subirá al servidor que aloja el proyecto, evitando conflictos. Por ello se llaman variables de entorno, porque nos permite variar su valor dependiendo de el entorno en el que estén.

Además, si consideramos que trabajamos en un proyecto de un banco que tiene varias las siguientes fases separadas por servidores independientes:

* Development (desarrollo)
* Production (producción)

Ahora supongamos que *Development* tiene una base de datos en MySQL que es sólo para hacer pruebas mientras se desarrolla y que *Production* tiene otra base de datos en otro servidor que es en la cual se guarda la información bancaria de nuestros clientes. Si no utilizamos variables de entorno, el código que funciona en *Development* no funcionaría en *Production,* ya que tendrían usuario y contraseña diferentes cada uno de los servidores para sus bases de datos. Así que tendríamos que cambiar las credenciales en nuestro código cada vez que quisiéramos pasar el código de *Development* a *Production*… algo que para nada es eficiente… Ahora imagina que tenemos 20 credenciales de autentificación diferentes para diferentes servicios en cada servidor y que, en vez de tener fases (es decir, dos servidores), tenemos seis… El problema sería realmente grave.

Pero gracias a las variables de entorno, podemos solucionar este problema. Bien, basta de tanto texto y pongámoslas en práctica con nuestra API de Node.js.

Como ya sabemos, debemos tener las siguientes dependencias instaladas para hacer funcionar nuestro proyecto:

*Express:* “***npm install express@4.16.2 --save”***

BodyParser: “***npm install body-parser@1.18.2 --save***”

Conector de MySQL: ***“npm install mysql@2.15.0 --save”***.

Y también será necesario instalar un nuevo paquete llamado *dotenv*, el cual nos permitirá utilizar variables de entorno de manera eficiente. Este paquete se intala con el siguiente comando: “**npm install dotenv@4.0.0 –save”**.

Después de instalarlo, es hora de ver cómo hemos implementado las variables de entorno en nuestro código previo. Crearemos un archivo con el nombre de *app.js* y utilizaremos el siguiente código en él:

app.js

"use strict";

**const** express = require('express');

**const** app = express();

**const** bodyParser = require('body-parser');

app.use(bodyParser.json());

**const** env = require('dotenv');

env.config();

**const** mysql = require('mysql');

**const** conexion = mysql.createConnection({

host: process.env.DB\_HOST,

user: process.env.DB\_USER,

password: process.env.DB\_PASSWORD,

port: process.env.DB\_PORT,

database: process.env.DB\_DATABASE

});

**const** portExpress = process.env.EXPRESS\_PORT;

**const** hostExpress = process.env.EXPRESS\_HOST;

conexion.connect(err **=>** {

if (err) {

console.log('Error trying to connect with Data Base: ' + err.stack);

throw err;

}

console.log("Conexión exitosa con la base de datos c:")

});

*//Función auxiliar para responderle al cliente*

**function** responderAlCliente( error, res, datos ) {

if ( error )

res.status(500).json(error);

else

res.status(200).json(datos);

}

*//Consulta general de todos los pendientes en la base de datos.*

app.get('/', (req, res) **=>** {

if ( conexion ) {

conexion.query('SELECT pendiente\_id, descripcion, estado FROM pendientes', (error, resultados) **=>** {

*//Si hay un error, le respondemos al cliente con el error.*

if (error){

return responderAlCliente( error, res );

}

*//Si no hay error, le respondemos al cliente con los*

*//resultados de la query.*

return responderAlCliente( null, res, resultados );

})

} else {

*//Si hay un error en la conexión, se lo indicamos al cliente.*

return responderAlCliente('Hubo un error con la conexión a MySQL :(', res)

}

})

*//Consultar cuando registros hay en la tabla de pendientes.*

.get('/count', (req, res) **=>** {

if ( conexion ) {

conexion.query('SELECT COUNT (pendiente\_id) AS numero\_de\_registros FROM pendientes', (error, contador) **=>** {

*//Si hay un error, le respondemos al cliente con el error.*

if (error){

return responderAlCliente( error, res );

}

*//Si no hay error, le respondemos al cliente con*

*//el contador de la query.*

return responderAlCliente( null, res, contador );

})

} else {

*//Si hay un error en la conexión, se lo indicamos al cliente.*

return responderAlCliente('Hubo un error con la conexión a MySQL :(', res)

}

})

*//Consultar cuando registros hay en la tabla de pendientes.*

.get('/siguienteIdAutoIncrementable', (req, res) **=>** {

if ( conexion ) {

conexion.query(`SELECT \`AUTO\_INCREMENT\` AS siguiente\_id\_autoincrementable

FROM INFORMATION\_SCHEMA.TABLES

WHERE TABLE\_SCHEMA = '${process.env.DB\_DATABASE}'

AND TABLE\_NAME = 'pendientes';`, (error, siguienteId) **=>** {

*//Si hay un error, le respondemos al cliente con el error.*

if (error){

return responderAlCliente( error, res );

}

*//Si no hay error, le respondemos al cliente con*

*//el siguienteId.*

return responderAlCliente( null, res, siguienteId );

})

} else {

*//Si hay un error en la conexión, se lo indicamos al cliente.*

return responderAlCliente('Hubo con error con la conexión a MySQL :(', res)

}

})

*//Consultar un pendiente por id.*

.get('/:idPendiente', (req, res) **=>** {

**const** idPendiente = req.params.idPendiente;

if ( conexion ) {

conexion.query(`SELECT pendiente\_id, descripcion, estado

FROM pendientes

WHERE pendiente\_id = ?`, [idPendiente], (error, pendiente) **=>** {

*//Si hay un error, le respondemos al cliente con el error.*

if (error){

return responderAlCliente( error, res );

}

*//Si no hay error, le respondemos al cliente con*

*//el pendiente retornado obtenido de la query.*

return responderAlCliente( null, res, pendiente );

})

} else {

*//Si hay un error en la conexión, se lo indicamos al cliente.*

return responderAlCliente('Hubo con error con la conexión a MySQL :(', res)

}

})

*//Agregar un nuevo pendiente a la base de datos.*

.post('/', (req, res) **=>** {

*/\**

*pendiente\_id lo mandamos como null para*

*que el AUTO\_INCREMENT de MySQL asigne*

*a pendiente\_id autoincrementalmente.*

*\*/*

**const** nuevoPendiente = {

pendiente\_id: null,

descripcion: req.body.descripcion,

estado: req.body.estado

}

if ( conexion ) {

conexion.query('INSERT INTO pendientes SET ?', [nuevoPendiente], (error, respuesta) **=>** {

if ( error ) {

*//Si hay un error, le respondemos al cliente con el error.*

return responderAlCliente( error, res );

}

*//Si no hay error, le respondemos al cliente con*

*//la repuesta retornada, obtenida de la query.*

return responderAlCliente( null, res, respuesta )

});

} else

return responderAlCliente('Hubo con error con la conexión a MySQL :(', res)

})

*//Modifica un pendiente existente en la base de datos.*

.put('/:idPendiente', (req, res) **=>** {

**const** idPendiente = req.params.idPendiente;

**const** pendientePorEditar = {

pendiente\_id: idPendiente,

descripcion: req.body.descripcion,

estado: req.body.estado

}

if ( conexion ) {

conexion.query(`UPDATE pendientes

SET descripcion = ?, estado = ?

WHERE pendiente\_id = ?`,

[pendientePorEditar.descripcion,

pendientePorEditar.estado,

pendientePorEditar.pendiente\_id], (error, respuesta) **=>** {

if ( error ) {

*//Si hay un error, le respondemos al cliente con el error.*

return responderAlCliente( error, res );

}

*//Si no hay error, le respondemos al cliente con*

*//la repuesta retornada, obtenida de la query.*

return responderAlCliente( null, res, respuesta )

});

} else

return responderAlCliente('Hubo con error con la conexión a MySQL :(', res)

})

*//Elimina un pendiente existente en la base de datos.*

.delete('/:idPendiente', (req, res) **=>** {

**const** idPendiente = req.params.idPendiente;

if ( conexion ) {

conexion.query(`DELETE FROM pendientes

WHERE pendiente\_id = ?`, [idPendiente], (error, respuesta) **=>** {

*//Si hay un error, le respondemos al cliente con el error.*

if (error){

return responderAlCliente( error, res );

}

*//Si no hay error, le respondemos al cliente con*

*//la respuesta retornada obtenido de la query.*

return responderAlCliente( null, res, respuesta );

})

} else {

*//Si hay un error en la conexión, se lo indicamos al cliente.*

return responderAlCliente('Hubo un error con la conexión a MySQL :(', res)

}

})

app.listen(portExpress, hostExpress, () **=>** {

console.log(`¡Escuchando en ${hostExpress}:${portExpress}/`);

});

Y necesitaremos crear un segundo archivo llamado “***.env***”, en el cual guardaremos todas las variables de entorno que hasta ahora teníamos:

*.env*

#Las líneas que empiezan con un '#', son tratadas

#como comentario y por lo tanto, son ignoradas.

#Configuración de MySQL

DB\_HOST=127.0.0.1

DB\_USER=root

DB\_PASSWORD=qwerty1234

DB\_PORT=3306

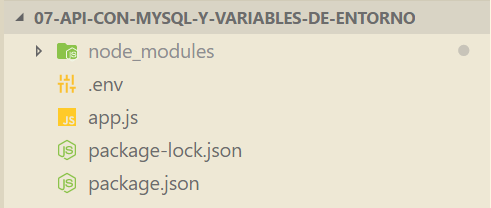
DB\_DATABASE=db\_pendientes

#Configuración de nuestro servidor

EXPRESS\_PORT=3000

EXPRESS\_HOST=127.0.0.1

Y la estructuración de nuestro proyecto, debería ser similar a la siguiente:



Bien, lo que ha cambiado solamente ha sido la primera sección de nuestra aplicación:

**const** env = require('dotenv');

env.config();

**const** mysql = require('mysql');

**const** conexion = mysql.createConnection({

host: process.env.DB\_HOST,

user: process.env.DB\_USER,

password: process.env.DB\_PASSWORD,

port: process.env.DB\_PORT,

database: process.env.DB\_DATABASE

});

**const** portExpress = process.env.EXPRESS\_PORT;

**const** hostExpress = process.env.EXPRESS\_HOST;

Como podremos ver, en vez de tener nuestras credenciales escritas directamente en el código de nuestra aplicación, ahora las tenemos en un archivo separado, llamado *“.env”,* el cual nos permite separar todas las credenciales de la lógica de nuestro servidor.

Para poder tener disponibles las variables de entorno de nuestro archivo “*.env*”, lo que tenemos que hacer es requerir el paquete *dotenv* en nuestro código:

**const** env = require('dotenv');

Y justo enseguida, ejecutar a su función llamada config():

env.config();

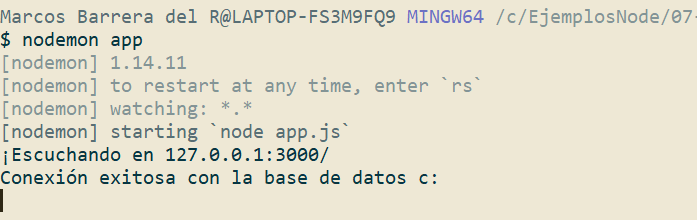
Desde el momento en el que ejecutamos la función config(), ya podremos utilizar todas la variables de entorno definidas en nuestro archivo *“.env”* a lo largo de nuestra aplicación.

Y para usarlas, utilizaremos un objeto de Node llamado ***process.env***. Por ejemplo, para proporcionar nuestra contraseña a la conexión de MySQL tan sólo hace falta:

password: process.env.DB\_PASSWORD,

Y como ***process.env*** ya tiene nuestra variable DB\_PASSWORD establecida en el archivo “***.env***”, ya podremos utilizar su valor.

Si probamos nuestra aplicación con ***nodemon app***, podremos ver que sigue funcionando perfectamente:



En resumen, al utilizar variables de entorno, aumentamos la seguridad, escalabilidad y eficiencia al estar desarrollando nuestra aplicación de Node.js.

***Nota:*** Si algún día se desea usar el controlador de versiones llamado Git, se recomienda encarecidamente que el archivo ***.env*** se coloque en el ***.gitignore*** de nuestro proyecto, para evitar que se suba al servidor remoto del proyecto.