**API en Express con CRUD Básico y MySQL**

Después de haber aprendido cómo se implementa una API con un CRUD Básico, es hora de implementar de una vez por todas una base de datos a nuestra aplicación. Nosotros elegimos MySQL.

Antes que nada, aunque esta guía no incluye la instalación de MySQL ni el manejo de bases de datos, nos menciona en dónde puede conseguir algún paquete de instalación que incluya MySQL.

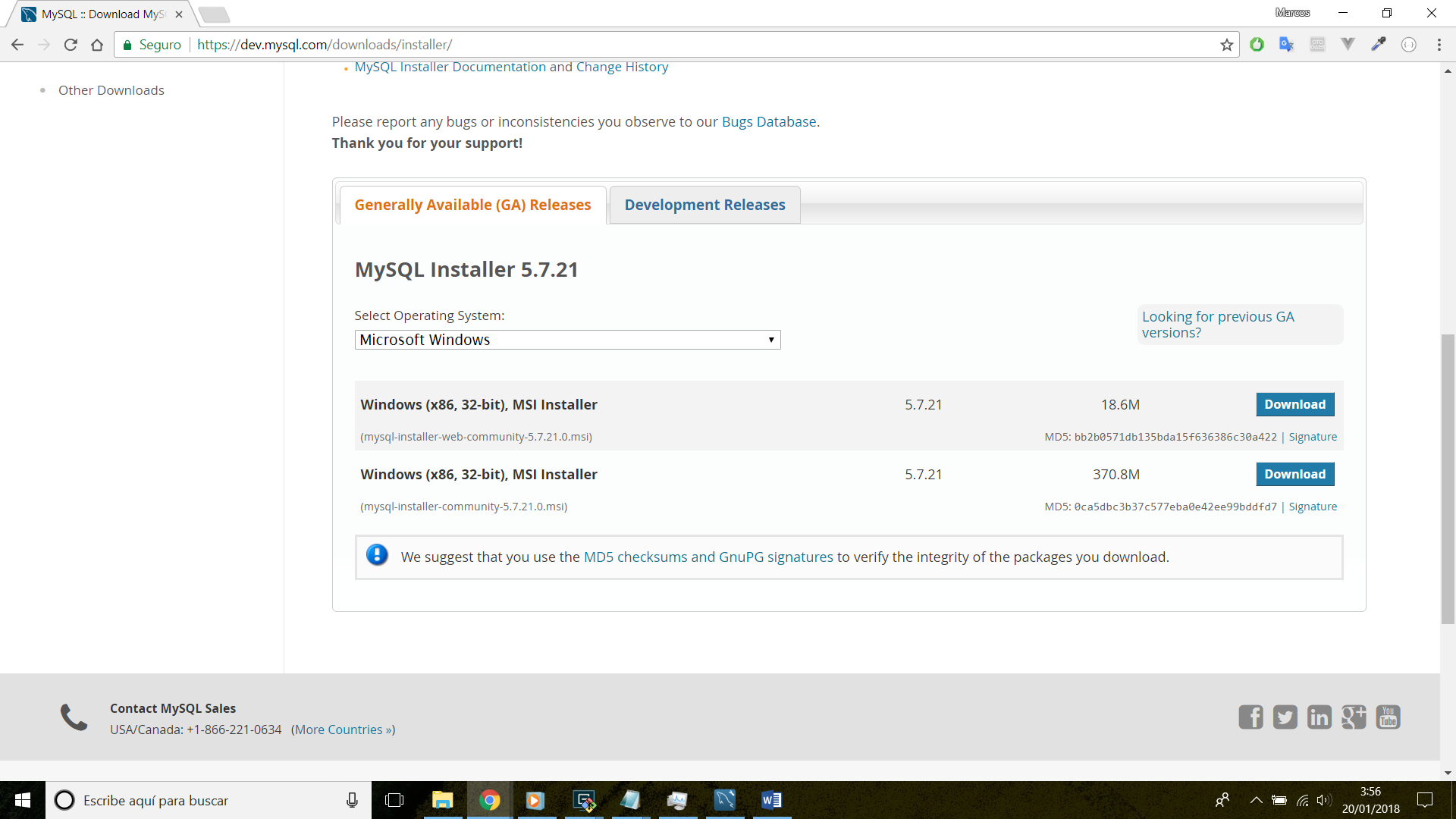
Un paquete muy bueno de fácil instalación en Windows que incluye Apache, ***MySQL*** y PHP es el llamado WAMP y lo puedes descargar desde la siguiente página:

<http://www.wampserver.com/en/>

Aunque nosotros preferimos el instalador que incluye MySQL Server y MySQL Workbench. Este último proporciona una interfaz de usuario que facilita el uso de MySQL y permite hacer todo tipo de configuraciones al mismo. MySQL Workbench será la herramienta que utilizaremos para la creación de nuestra base de datos de ejemplo.

Puedes descargar MySQL Server y MySQL Workbench desde la página oficial de descargas de MySQL:

<https://dev.mysql.com/downloads/installer/>



Debes seleccionar el instalador correspondiente a tu equipo y una vez descargado, seguir las indicaciones del instalador. La instalación que nosotros realizamos fue una “*Custom*” o “*Personalizada*” la cual nos permite seleccionar sólo los componentes que queramos. En nuestro caso, solo instalamos MySQL Server y MySQL Workbench, pero si quiere una configuración estándar, puedes elegir la opción de instalación por defecto.

Bien, una vez que tengas instalado MySQL, accede a el desde la interfaz gráfica que hayas elegido y ejecuta la siguiente *query:*

CREATE DATABASE db\_pendientes;

USE db\_pendientes;

SET default\_storage\_engine=INNODB;

CREATE TABLE pendientes (

pendiente\_id INT NOT NULL AUTO\_INCREMENT,

descripcion VARCHAR(1000),

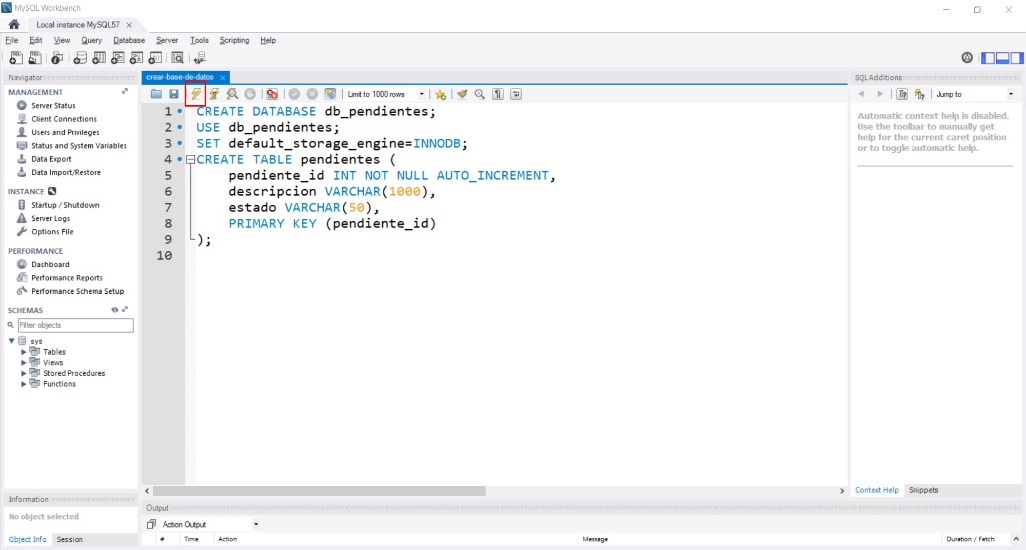
estado VARCHAR(50),

PRIMARY KEY (pendiente\_id)

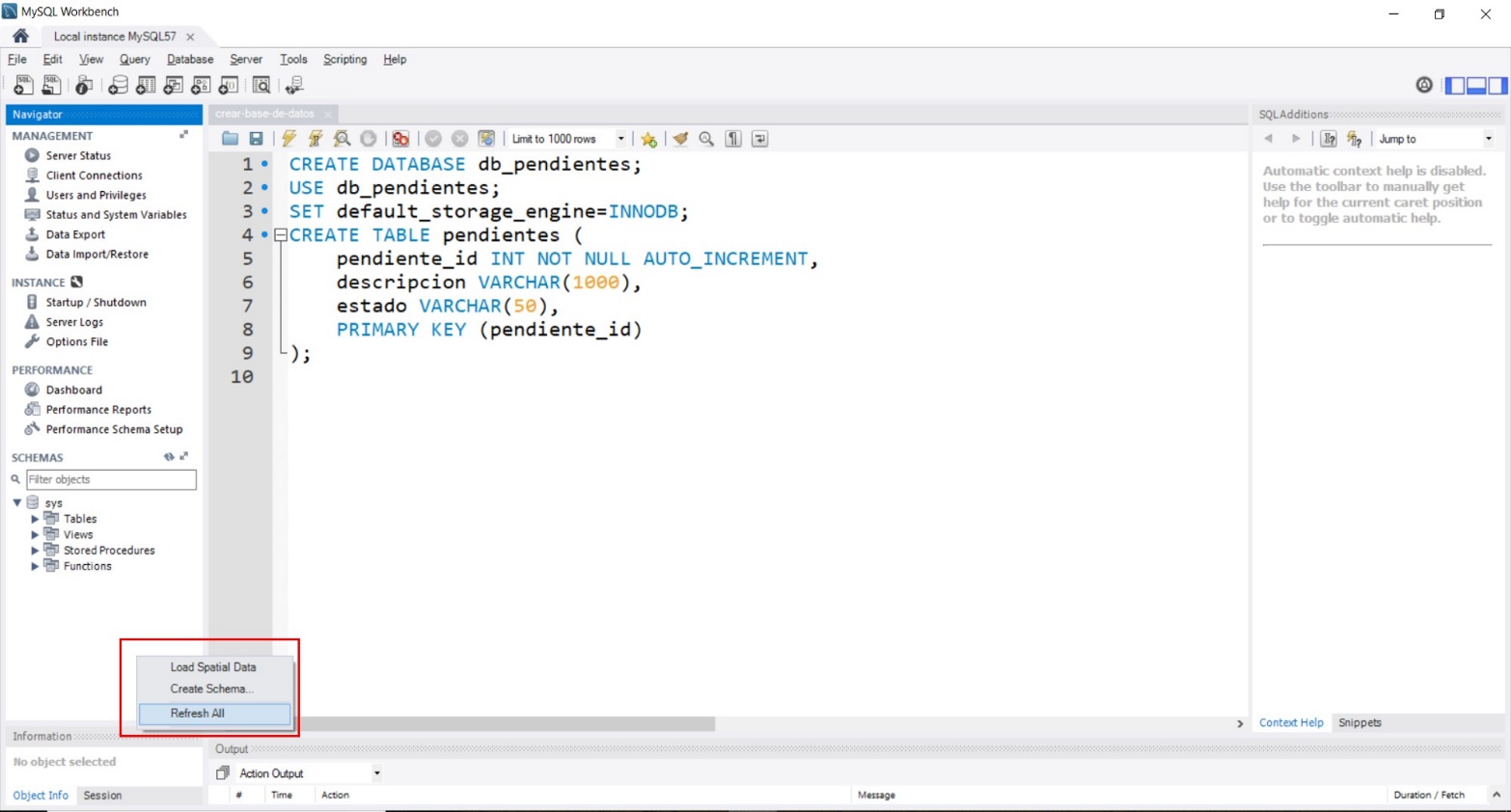
);

**NOTA:** La instrucción “SET default\_storage\_engine=INNODB;” nos permitirá más adelante implementar transacciones con MySQL.

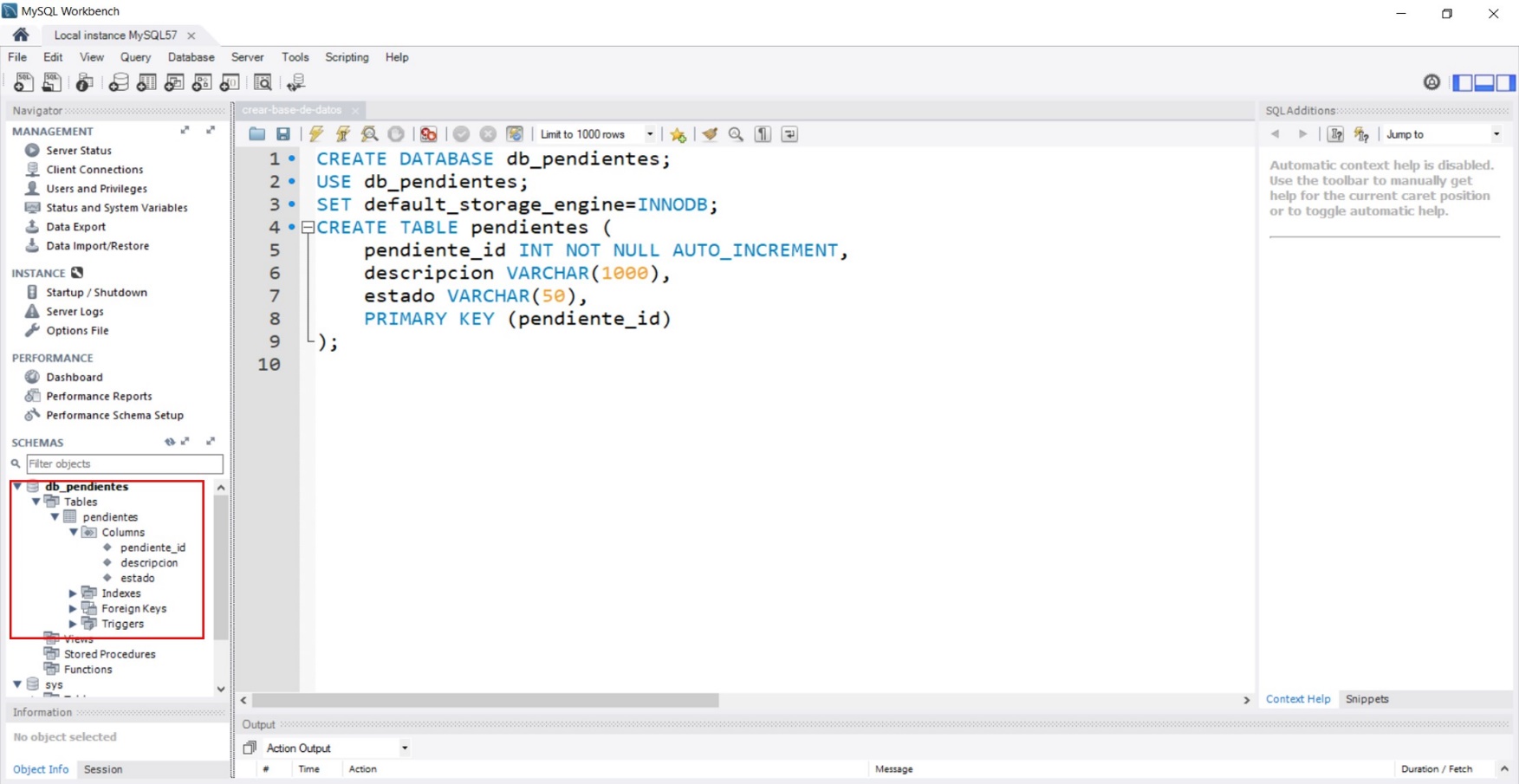
En MySQL Workbench se ejecuta donde indica el recuadro rojo:



Después de haber ejecutado la query, refrescaremos la como indica la siguiente página:



Y finalmente podremos ver que nuestra base de datos se ha creado satisfactoriamente:



Ahora sí podemos proseguir a ver cómo está construida nuestra API y cómo implementa la conexión con MySQL desde Node.js.

Además de instalar *Express* con el comando “***npm install express@4.16.2 --save”*** y de instalar Body Parser, con “***npm install body-parser@1.18.2 --save***”***,*** esta vez tendremos que instalar el *package* llamado *mysql*, y este se instalar con el comando ***“npm install mysql@2.15.0 --save”***.

Una que tengamos instalado *Express,* el package manejador de *mysql* para Nodey *body-parser*, crearemos un archivo llamado “*app.js”*. El código completo de la aplicación es el siguiente:

"use strict";

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

**const** app = express();

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

app.use(bodyParser.json());

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

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

host: '127.0.0.1',

user: 'root',

password: 'qwerty1234',

port: 3306,

database: 'db\_pendientes'

});

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 = 'db\_pendientes'

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 un 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 un 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)

})

.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)

}

})

**const** portExpress = 3000;

**const** hostExpress = '127.0.0.1';

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

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

});

El primer cambio que nos encontramos es que, en lugar de tener nuestros registros en memoria, utilizamos una base de datos en MySQL:

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

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

host: '127.0.0.1',

user: 'root',

password: 'qwerty1234',

port: 3306,

database: 'db\_pendientes'

});

Lo primero que hacemos es importar el *package* “*mysql”* para poder interactuar con nuestra base de datos en MySQL. Después creamos una constante llamada *conexion*, la cual instanciaremos con mysql.createConnection({}).

El parámetro que le pasamos a createConnection es un objeto que contiene distintas propiedades, las cuales explicaremos enseguida:

***host:*** El host en el que esté alojada nuestra base de datos, puede ser un servidor externo, pero en nuestro caso, como es nuestra propia computadora localmente, utilizamos *“127.0.0.1”,* aunque también podría ser utilizado “*localhost*”.

***user:*** Algún usuario de la base de datos que tenga privilegios de lectura y escritura, en nuestro caso, apenas instalamos MySQL, así que el usuario por defecto es “*root*”.

***password:*** La contraseña del usuario que hayamos elegido utilizar.

***port:*** El puerto por reservado para MySQL, el puerto por defecto después de una instalación de MySQL es 3306.

***database:*** Una base de datos en MySQL con la que queramos establecer una conexión.

Además de estas propiedades, existen otras varias que nos permiten realizar un manejo más especifico de MySQL desde Node, pero no profundizaremos en ellas, ya que no son necesarias por el momento.

Después, para realizar la conexión con MySQL, mandamos a llamar connect(...) desde la constante conexión que acabamos de crear anteriormente.

El parámetro que recibe conexión.connect( ), es una *callback:*

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:")

});

Dicha *callback,* recibe como parámetro un error, en caso de que haya ocurrido un error mientras conexion.connect() intenta hacer conexión con la base de datos. Comprobamos con una sentencia ***if***si existe un error ( if (err) ). en caso de haberlo, entonces se imprime el error en consola y se arroja con “throw err;”. En caso de que no haya error, imprimimos en la consola que la conexión fue exitosa.

El siguiente cambio que implementamos, fue hacer una *función auxiliar* que utilizaremos a lo largo de todos los *endpoints,* para responderle al cliente, ya sea con los resultados de las *queries* o con los errores que puedan llegar a presentarse:

*//Función auxiliar para responderle al cliente*

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

if ( error )

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

else

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

}

La función responderAlCliente( error, res, datos ), como podemos ver, recibe como primer parámetro ***error****,* en caso de haberlo*.* Como segundo parámetro, recibe ***res***, el objeto de *Express* para mandar una respuesta (*response*) al cliente, y como tercer parámetro, recibe ***datos****.*

Después, con una simple sentencia “if ( error )”, comprobamos si hay algún error como parámetro. En caso de que sí haya un error, le responderemos con dicho error al cliente con “res.status(500).json(error);”. En caso de que haya ningún error, lo que se ejecutará, será lo que viene en la sentencia else, es decir, la respuesta con los datos de las *queries* con “res.status(200).json(datos);”.

No te preocupes si no entiendes completamente como encaja esta *función auxiliar* en la aplicación. Más adelante veremos cómo sacarle provecho.

Enseguida, podremos ver que nuestra consulta general a cambiado sustancialmente:

*//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)

}

})

Lo primero que hacemos al llegar a la consulta general es comprobar si hay conexión con MySQL con un ***if***: “if ( conexion )”. Si no hay *conexión*, le responderemos al cliente con nuestra *función auxiliar:*

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

Como vimos anteriormente, esta función como primer parámetro acepta un error (nuestro mensaje, en este caso), como segundo el objeto ***res*** de *Express* ycomo tercero, recibe un parámetro de *datos,* pero como queremos mandar un error al cliente, no es necesario proporcionar el tercer parámetro.

En caso de que si haya conexión, mandamos a llamar a conexion.query(...) que recibe como primer parámetro una *query* en lenguaje *SQL:*

'SELECT pendiente\_id, descripcion, estado FROM pendientes'

Como podemos ver, es una simple *query* que consulta todos los registros de nuestra tabla *pendientes* y que solicita explícitamente las tres columnas que tenemos. Otra manera de solicitar todas las columnas en una sentencia SQL podría haber sido: 'SELECT \* FROM pendientes'

Pero se considera una manera ineficiente de realizar *queries* en bases de datos y por lo tanto se podría decir que es una mala práctica, por lo que preferimos utilizar la primera forma siempre que sea posible.

Después de especificar la *query* como primer parámetro de conexion.query, especificamos el segundo parámetro, el cual es una *callback:*

(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.*

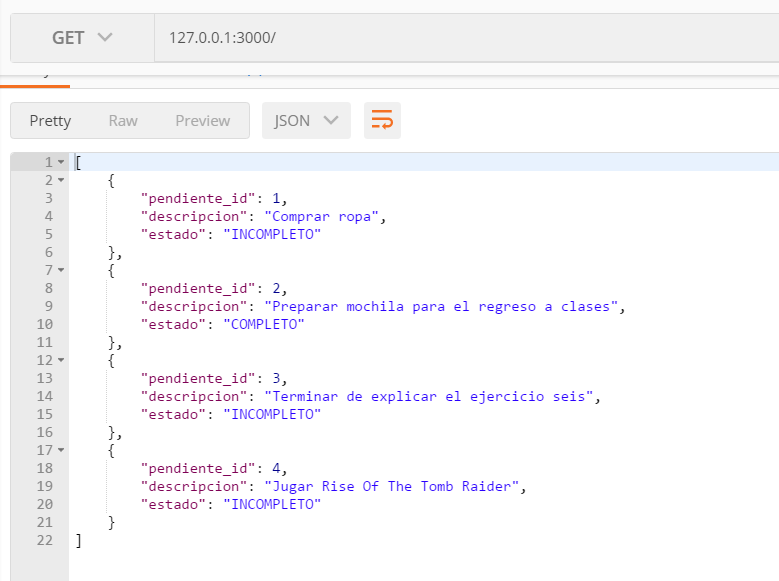
return responderAlCliente( null, res, resultados );

}

Dicha *callback* recibe como parámetros a ***error***, en caso de que haya alguno y los *resultados* de nuestra *query*. Y dentro de la *callback* comprobamos si en verdad hay un ***error*** con una sentencia “if (error)”.

En caso de que se haya producido un error al haber ejecutado la *query*, retornaremos la función responderAlCliente(error, res) que explicamos anteriormente, y en este caso, le pasaremos como primer parámetro el ***error*** y como segundo, el objeto ***res*** que *Express* nos provee en todos nuestros *endpoints* para poder responderle al cliente*.*

Para comprobar la funcionalidad de nuestra consulta general, ejecutaremos nuestro servidor con ***nodemon app***. Previamente insertamos registros para tener que consultar. Así que una vez que el servidor este en marcha, si hacemos una petición a ***“***127.0.0.1:3000/***”*** con el método GET, podremos ver que nuestra consulta general funciona apropiadamente:



El siguiente *enpoint* (GET), es un simple contador que nos responde con los registros actuales en nuestra tabla *pendientes:*

*//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)

}

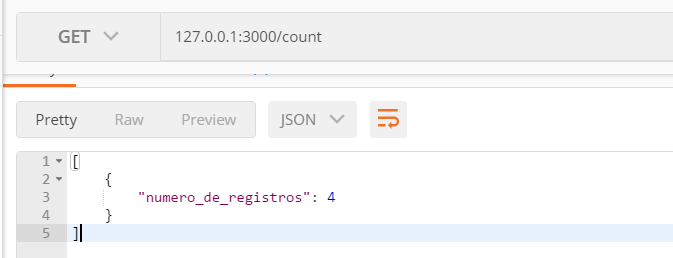
})

Al igual que en la consulta general, consultamos si hay *conexión* con un if y si no la hay, respondemos con un mensaje de error al cliente.

En caso de que haya conexión, ejecutamos “conexion.query(...)” y como primer parámetro le pasamos la siguiente *query:*

'SELECT COUNT (pendiente\_id) AS numero\_de\_registros FROM pendientes'

Una sencilla *query* que cuenta los registros que tenemos en nuestra tabla de *pendientes* y como resultado nos responde con un array de longitud *1* y dentro del array, un objeto que tiene como propiedad “numero\_de\_registros” y como valor de esa propiedad, el número de registros de la tabla. Si hacemos una petición GET a la ruta con “*127.0.0.1/count*” podremos comprobarlo:



***Nota:*** En la gran mayoría de *queries* que se realicen con el *package* de *mysql*, los resultados vendrán dentro de un array como en el ejemplo anterior.

Después como segundo parámetro de conexion.query(), utilizamos una *callback,* y al igual que en el *enpoint* anterior, verificamos que no se haya producido ningún error. En caso de que lo haya, respondemos con el error al cliente. En el caso de que no haya ningún error, le respondemos al cliente con el contador.

(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 );

}

Continuaremos con el siguiente *endpoint,* el cual nos permite saber cuál es el siguiente ID *autoincrementable* de nuestra tabla de *pendientes.* Este tipo de consulta suele ser útil en los casos en donde permitamos que se borren registros de nuestras tablas con ID *autoincrementable* y después queramos crear más registros desde algún cliente o cuando queremos hacer traspasos a bases de datos históricas:

*//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 = 'db\_pendientes'

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

}

})

En este caso, nuestro *endpoint* será de tipo GET y accederemos a el mediante la ruta '/siguienteIdAutoIncrementable'.

Nuevamente, comprobamos que haya *conexión*, si no la hay, mandamos un error al cliente y si sí hay conexión, procedemos a ejecutar una *query* de SQL con conexion.query.

La *query* en cuestión es:

`SELECT \`AUTO\_INCREMENT\` AS siguiente\_id\_autoincrementable

FROM INFORMATION\_SCHEMA.TABLES

WHERE TABLE\_SCHEMA = 'db\_pendientes'

AND TABLE\_NAME = 'pendientes';`

***Nota:*** Para una mejor legibilidad utilizamos String Literals de JavaScript.

Este tipo de *queries* no suele ser usual. A grandes rasgos, lo que hace es obtener el siguiente valor *autoincrementable* del ID de la tabla *pendientes,* basándose en los esquemas de la base de datos *db\_pendientes*.

Y en la *callback* comprobamos y hay error, en caso de haber error, respondemos con “return responderAlCliente( error, res );”. En caso de que el error sea *null* o *undefined,* respondemos con “return responderAlCliente( null, res, siguienteId );”.

Para mostrar el funcionamiento de esta *query*, borramos el *pendiente* con ID: *4*



Y al consultar nuestro *endpoint:* “*127.0.0.1:3000/siguienteIdAutoIncrementable”* nos responderá con lo siguiente.



Y como respuesta obtendremos un array que contiene un objeto con la propiedad de siguiente\_id\_autoincrementable, la misma que especificamos en la *query* y como valor de dicha propiedad, el siguiente ID *autoincrementable.* Aun cuando borramos el *pendiente* con el ID: *4,* esta *query* nos responde con el siguiente ID *autoincrementable,* es decir, ID: 5.

El siguiente *endpoint,* es nuestra consulta individual que ya hemos implementado en anteriores ejercicios, pero con la diferencia obvia de que utiliza MySQL.

*//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 un error con la conexión a MySQL :(', res)

}

})

Como ya habíamos explicado en anteriores ejercicios, obtendremos el ID del *pendiente* querido con:

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

Nuevamente comprobamos que haya conexión, y ejecutamos una *query* con conexion.query. La *query* a ejecutar es la siguiente:

`SELECT pendiente\_id, descripcion, estado ç

FROM pendientes

WHERE pendiente\_id = ?`

Como podemos ver, es una consulta simple que nos regresa los campos de un *pendiente* con el ido dado. Pero si miramos con atención, notaremos que hay un “?”, en lugar de un ID. Este “?” nos indica que en dicha posición se colocará el primero valor del array que pasamos como segundo parámetro a conexion.query

Por ejemplo, si nosotros hiciéramos una *request* a este *endpoint* de consulta individual y pasáramos el ID: ***4*** como parámetro, al ejecutar:

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

FROM pendientes

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

Lo que pasaría es que internamente el *package* de *mysql* filtraría los parámetros que le pongamos en el array que va como segundo parámetro ([idPendiente] en nuestro caso) y una vez filtrados, reemplazaría el “?” que está en la *query* con el elemento del array que le pasamos, es decir, idPendiente con el valor de ***4***, el cual mandamos en nuestra *request.*

Por lo tanto, si nosotros mandamos el ID: ***4***, se ejecutaría esto inicialmente:

`SELECT pendiente\_id, descripcion, estado

FROM pendientes

WHERE pendiente\_id = ?`, [idPendiente]

Y el paquete de *mysql* internamente ejecutaría esto:

`SELECT pendiente\_id, descripcion, estado

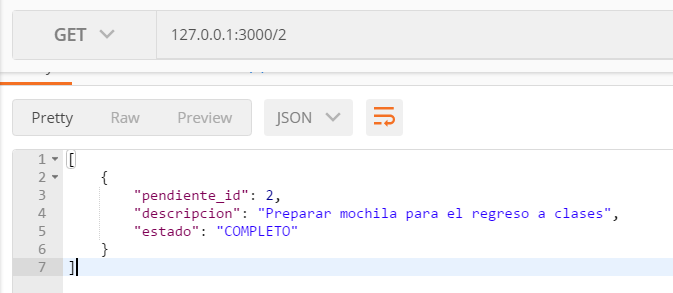
FROM pendientes

WHERE pendiente\_id = '4' `

Esto se hace con la finalidad de filtrar posibles inyecciones SQL que puedan perjudicar a nuestra base de datos o incluso borrarla por completo. Por ello, se recomienda siempre utilizar este método al hacer *queries.* Ya que, si no utilizamos este método, correríamos el riesgo de que algún atacante envíe como parámetro un “1; DROP TABLE pendientes;”, por ejemplo y elimine nuestra tabla.

Por último*,* al igual que en los otros *enpoints,* verificamos si hay error o no y le respondemos al cliente con el error o con el resultado, según sea el caso.

Y al probar esta consulta individual con el *pendiente* de ID: ***2***, nos arroja el resultado deseado:



Es hora de explicar cómo funciona el *endpoint* para agregar más *pendientes:*

*//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)

})

Primero extraemos la *descripción (tipo string)* y el *estado (tipo string)* del cuerpo (*body*)de la *request* y procedemos a construir un objeto de nuevoPendiente:

*/\**

*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

}

Como lo menciona el comentario del código, pendiente\_id se manda como *null,* ya que el ID lo colocará automáticamente MySQL por ser *autoincremental*.

La “descripcion” la obtenemos con “req.body.descripcion” y “estado” con “req.body.estado”.

Después comprobaremos si hay conexión, como es habitual. Finalmente ejecutaremos lo siguiente:

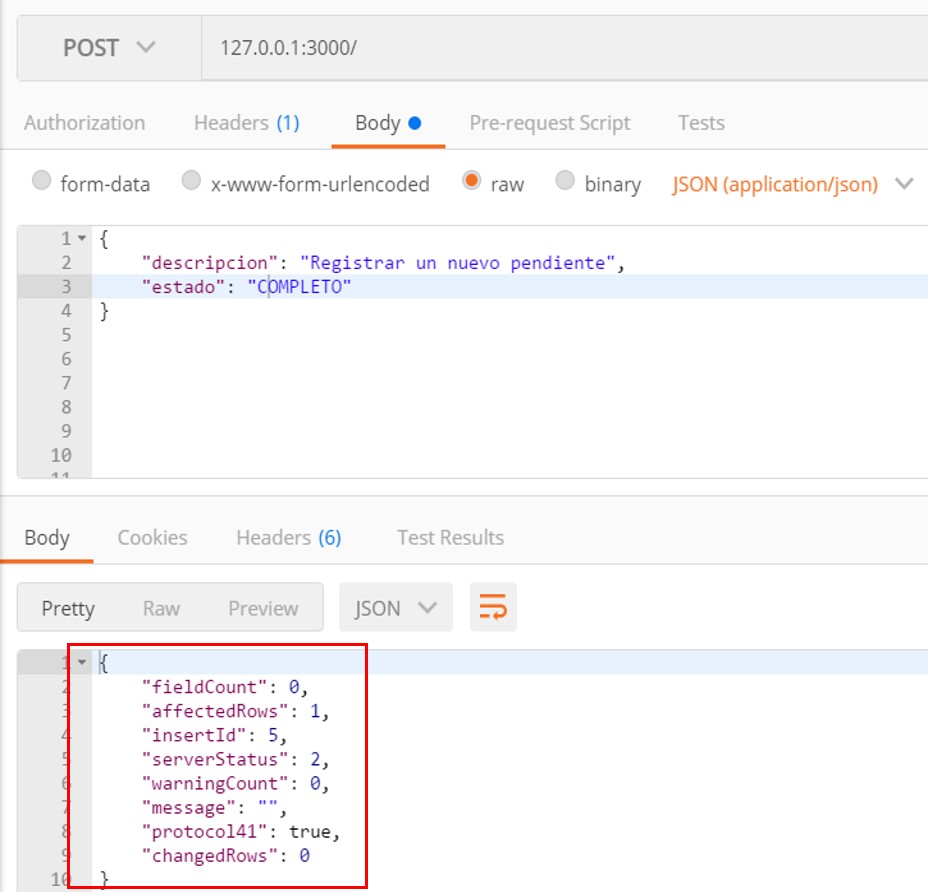
conexion.query('INSERT INTO pendientes SET ?', [nuevoPendiente],(error, respuesta)**=>**{})

Esta *query* es suficiente para insertar un nuevo registro en el sistema, de esta manera evitamos escribir la *query* con la sintaxis SQL: *VALUES*. Básicamente lo que hace el paquete de *mysql* al recibir *queries* de este estilo es convertirlas al formato típico automáticamente:

`INSERT INTO pendientes (pendiente\_id, descripcion, estado)

VALUES (0, ‘midescripcion’, 'miestado')`

Al probar el registro de tipo POST, nos responde con lo siguiente:



Esa es la respuesta que nos da MySQL al dar registrar un nuevo registro en la base de datos.

Después de explicar el *endpoint* para registrar un *pendiente* (con POST), es hora de explicar el *endpoint de* modificación (con PUT).

.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)

})

Al igual que en el registro de un nuevo *pendiente*, creamos un objeto, en este caso se llama “pendientePorEditar”, pero esta vez, en lugar de mandar un “pendiente\_id” con valor de *null*, mandamos el “idPendiente”, el cual obtuvimos desde el parámetro de la URL con “req.params.idPendiente”.

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

**const** pendientePorEditar = {

pendiente\_id: idPendiente,

descripcion: req.body.descripcion,

estado: req.body.estado

}

Después comprobamos si hay conexión con un ***if***. Y procedemos a ejecutar:

conexion.query(`UPDATE pendientes

SET descripcion = ?, estado = ?

WHERE pendiente\_id = ?`,

[pendientePorEditar.descripcion,

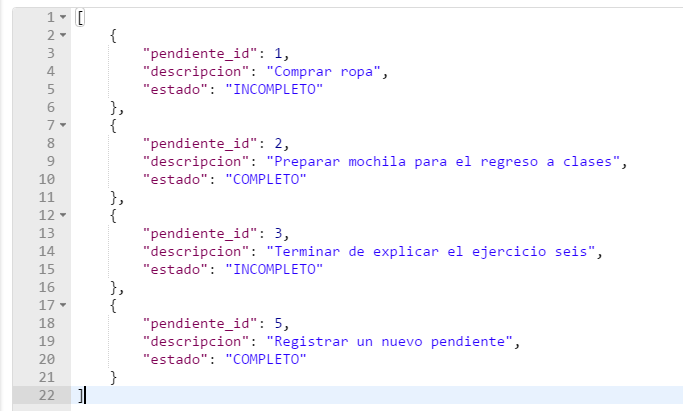
pendientePorEditar.estado,

pendientePorEditar.pendiente\_id],

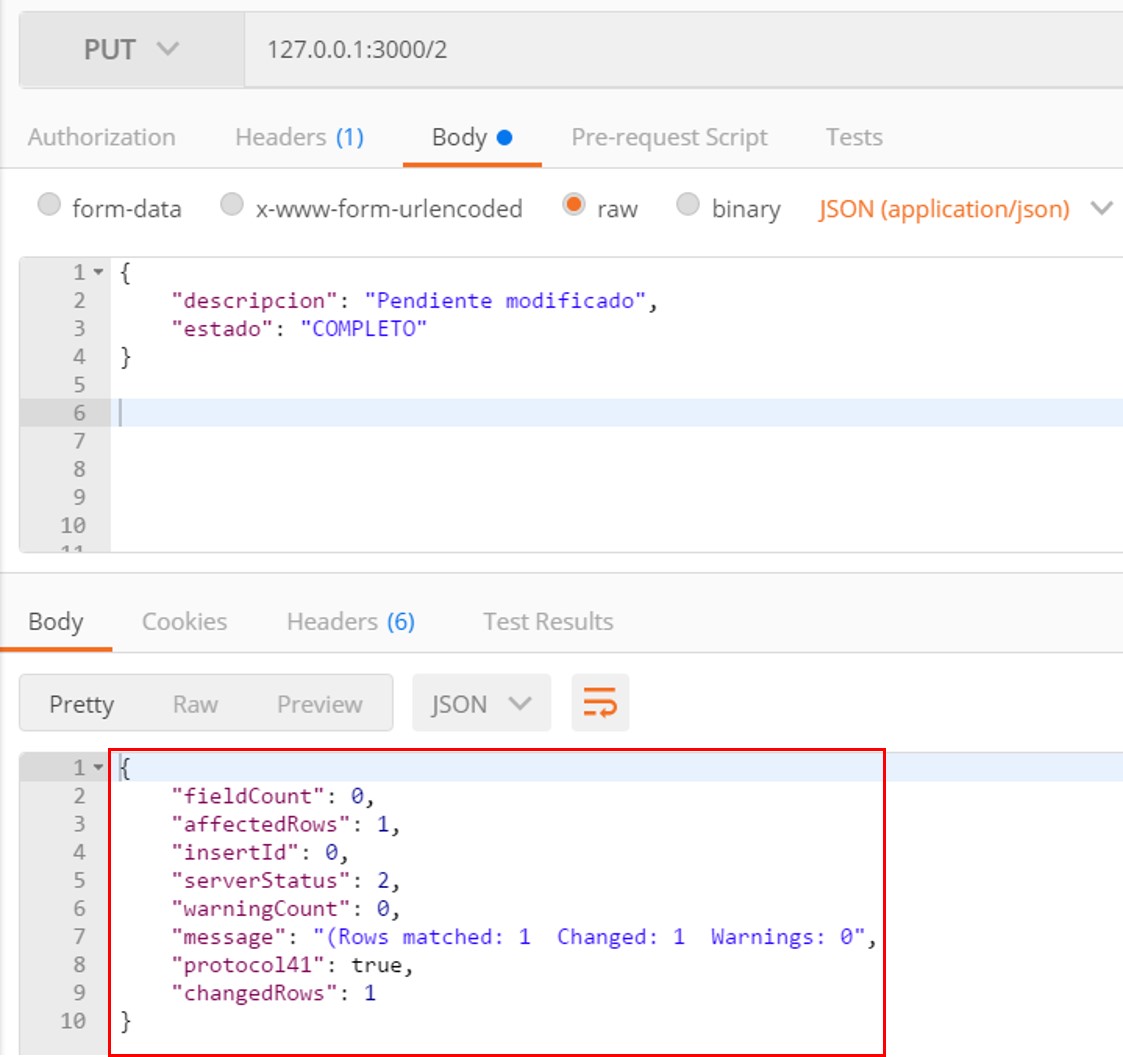
Como ya vimos anteriormente, el paquete de *mysql* reemplazará todos los “?” que haya en la *query* con los valores que contenga el array en el orden en el los proporcionemos. Por ello, tenemos que tener especial cuidado a la hora de acomodar los valores en el array.

En este caso el valor de “pendientePorEditar.descripcion” reemplazará el “?” de “descripcion = ?”, el valor de “pendientePorEditar.estado” reemplazará el “?” de “estado = ?” y por último, el valor de “pendientePorEditar.pendiente\_id” reemplazará el “?” de “pendiente\_id = ?”.

Antes de probar el *endpoint* de modificación, veremos cuáles son los *pendientes* actuales:

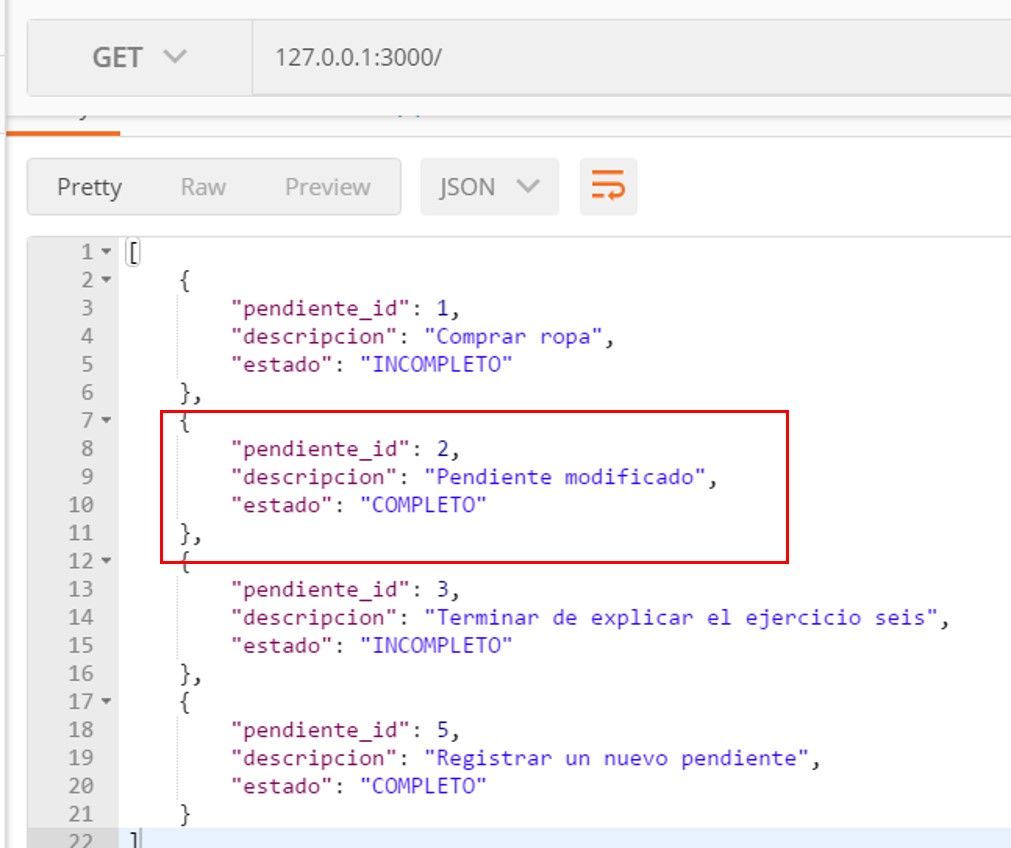


Ahora modificamos el *pendiente* con ID: ***2***:



El *endpoint* nos responde con este mensaje. Lo que significa que todos se ejecutó correctamente.

Y comprobamos que el *pendiente* con ID: ***2*** ha sido modificado correctamente:



Y finalmente, veremos cómo está estructurado el *endpoint* de eliminación (con DELETE):

*//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)

}

})

Primero, obtenemos el idPendiente desde el parámetro de la ruta.

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

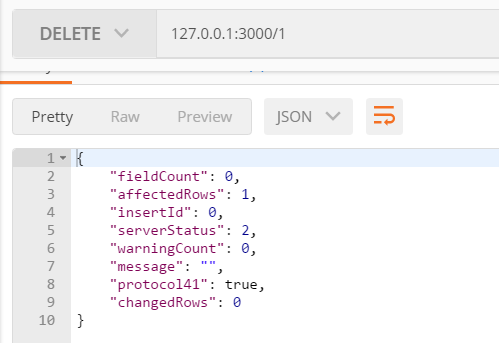
Y si hay una conexión con la base de datos, ejecutamos:

conexion.query(`DELETE FROM pendientes

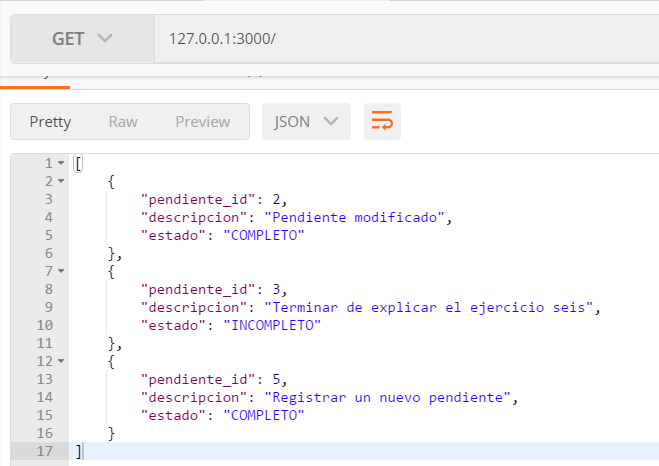
WHERE pendiente\_id = ?`, [idPendiente]

Y como ya lo sabes, el “?” será reemplazado por el valor de idPendiente.

Procedemos a eliminar un *pendiente,* por ejemplo, el que tiene el ID: ***1:***



MySQL nos responde con lo de la imagen anterior. Por último, comprobemos que se haya borrado:



Y efectivamente… el *pendiente* con ID: ***1***, ha sido removido.