**Implementación de modelos con Express**

Ahora que ya separamos la lógica de las rutas, de la lógica de la inicialización de servidor en los archivos de ***pendiente.routes.js*** y ***app.js***, aún nos queda separar el código de las rutas, del código que se encarga de hacer las *queries*. Si ahora mismo vemos una de las rutas que tenemos para los pendientes, por ejemplo, la consulta individual:

*//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) **=>** {

if (error)

return responderAlCliente( error, res );

return responderAlCliente( null, res, pendiente );

})

} else

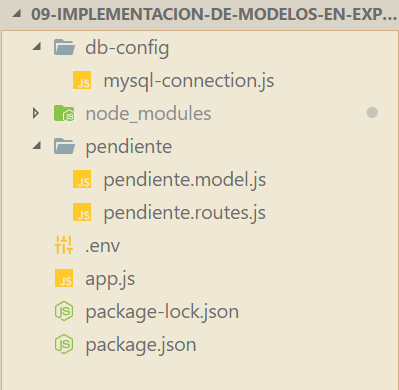
return responderAlCliente('Hubo con error con la conexión a MySQL :(', res);

})

Nos percataremos de que el código o la lógica del manejo de las rutas está mezclado con el código que se encarga de manejar las *queries* de MySQL. Arreglémoslo.

A continuación, te mostraremos la estructura y los cambios que aplicamos para poder implementar Modelos en *Express:*

*Estructura del proyecto:*



El archivo “*app.js”* quedó intacto:

*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** pendiente = require('./pendiente/pendiente.routes');

app.use('/pendiente', pendiente);

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

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

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

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

});

El archivo “.*env”* tampoco sufrió cambios:

*.env*

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

**#como comentario y por lo tanto, son ignoradas.**

**#NO SE RECOMIENDA SUBIR EL ARCHIVO .env A REPOSITORIOS**

**#DE GIT, PERO POR CUESTIONES PRÁCTICAS, LO SUBIREMOS ESTA VEZ**

**#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**

El archivo de la conexión queda intacto también:

*db-config/mysql-connection.js*

**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

});

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:")

});

module.exports = conexion;

Lo que si notarás es que hay un nuevo archivo llamada *pendiente.model.js* y este archivo contiene lo siguiente:

*pendiente/pendiente.model.js*

**const** conexion = require('../db-config/mysql-connection');

**let** Pendiente = { };

Pendiente.obtenerTodos = ( res, cb ) **=>** {

if ( conexion ) {

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

if (error)

return cb( error, res );

return cb( null, res, resultados );

})

} else

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

}

Pendiente.contador = ( res, cb ) **=>** {

if ( conexion ) {

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

if ( error )

return cb( error, res );

return cb( null, res, contador );

})

} else

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

}

Pendiente.siguienteId = ( res, cb ) **=>** {

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) **=>** {

if (error)

return cb( error, res );

return cb( null, res, siguienteId );

})

} else

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

}

Pendiente.encontrarPorId = ( idPendiente, res, cb ) **=>** {

if ( conexion ) {

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

FROM pendientes

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

if (error)

return cb( error, res );

return cb( null, res, pendiente );

})

} else

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

}

Pendiente.insertar = ( nuevoPendiente, res, cb ) **=>** {

if ( conexion ) {

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

if ( error )

return cb( error, res );

return cb( null, res, respuesta )

});

} else

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

}

Pendiente.actualizar = ( pendientePorEditar, res, cb ) **=>** {

if ( conexion ) {

conexion.query(`UPDATE pendientes

SET descripcion = ?, estado = ?

WHERE pendiente\_id = ?`,

[pendientePorEditar.descripcion,

pendientePorEditar.estado,

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

if ( error )

return cb( error, res );

return cb( null, res, respuesta )

});

} else

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

}

Pendiente.eliminar = ( idPendiente, res, cb ) **=>** {

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 cb( error, res );

return cb( null, res, respuesta );

})

} else

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

}

Pendiente.responderAlCliente = ( error, res, datos ) **=>** {

if ( error )

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

else

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

}

module.exports = Pendiente;

Como podrás ver, ahora toda nuestra lógica que maneja las *queries* de MySQL se encuentra en este archivo. Lo primero que hacemos es importar el archivo de la conexión de MySQL

**const** conexion = require('../db-config/mysql-connection');

Y justo después creamos un objeto llamado *Pendiente* que contendrá cada uno de los métodos/funciones que consultan, modifican o eliminan en la base de datos.

**let** Pendiente = { };

Cómo podemos ver, ahora existe un método para cada una de las funcionalidades previas que teníamos para conectarnos con MySQL.

*/\* Obtiene todos los registros de la tabla pendientes y recibe 'res'*

*y una callback llamada 'cb'.*

*\*/*

Pendiente.obtenerTodos = ( res, cb ) **=>** {

*//Código*

}

*/\* Obtiene cuántos registros hay en la tabla de pendientes.*

*Recibe 'res' y una callback llamada 'cb'.*

*\*/*

Pendiente.contador = ( res, cb ) **=>** {

*//Código*

}

*/\* Obtiene el siguiente ID autoincrementable en la tabla de pendientes.*

*Recibe 'res' y una callback llamada 'cb'.*

*\*/*

Pendiente.siguienteId = ( res, cb ) **=>** {

*//Código*

}

*/\* Obtiene el registro de la tabla pendientes que coincida con el*

*ID que se pase como parámetro 'idPendiente'.*

*Recibe 'idPendiente' que es el ID del pendiente que se desea encontrar.*

*También recibe 'res' y una callback llamada 'cb'*

*\*/*

Pendiente.encontrarPorId = ( idPendiente, res, cb ) **=>** { */\*Código\*/* }

*/\* Inserta un nuevoPendiente en la tabla de pendientes.*

*Recibe como parámetro 'nuevoPendiente', el cual debe*

*de contener las propiedades de un pendiente.*

*También recibe 'res' y una callback llamada 'cb'.*

*\*/*

Pendiente.insertar = ( nuevoPendiente, res, cb ) **=>** {

*//Código*

}

*/\* Edita un pendiente en la tabla de pendientes.*

*Recibe como parámetro 'pendientePorEditar', el cual debe*

*de contener las propiedades editadas de un pendiente existente.*

*También recibe 'res' y una callback llamada 'cb'.*

*\*/*

Pendiente.actualizar = ( pendientePorEditar, res, cb ) **=>** {

*//Código*

}

*/\* Elimina el registro de la tabla pendientes que coincida con el*

*ID que se pase como parámetro 'idPendiente'.*

*Recibe 'idPendiente' que es el ID del pendiente que se desea eliminar.*

*También recibe 'res' y una callback llamada 'cb'*

*\*/*

Pendiente.eliminar = ( idPendiente, res, cb ) **=>** {

*//Código*

}

*/\* Se encarga de responderle al cliente, ya sea con un error*

*o con los resultados de una query previamente ejecutada.*

*Recibe 'error' en caso de haberlo como primer parámetro o null.*

*Recibe 'res', el objeto de respuest de Express como segundo parámetro.*

*Recibe 'datos', en caso de haberlos como tercer parámetro o null.*

*\*/*

Pendiente.responderAlCliente = ( error, res, datos ) **=>** {

*//Código*

}

Y finalmente exportamos el objeto de *Pendiente* para poder importarlo en otros archivos/módulos:

module.exports = Pendiente;

Bien, ahora es tiempo de ver qué cambios implementamos en el archivo de *pendiente.routes.js:*

*pendiente/pendientes.routes.js*

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

**const** router = express.Router();

*//Importamos el modelo de Pendiente.*

**const** Pendiente = require('./pendiente.model');

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

router

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

return Pendiente.obtenerTodos( res, Pendiente.responderAlCliente );

})

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

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

return Pendiente.contador( res, Pendiente.responderAlCliente );

})

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

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

return Pendiente.siguienteId( res, Pendiente.responderAlCliente );

})

*//Consultar un pendiente por id.*

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

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

return Pendiente.encontrarPorId( idPendiente, res, Pendiente.responderAlCliente );

})

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

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

**const** nuevoPendiente = {

pendiente\_id: null,

descripcion: req.body.descripcion,

estado: req.body.estado

}

return Pendiente.insertar( nuevoPendiente, res, Pendiente.responderAlCliente );

})

*//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

}

return Pendiente.actualizar( pendientePorEditar, res, Pendiente.responderAlCliente );

})

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

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

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

return Pendiente.eliminar( idPendiente, res, Pendiente.responderAlCliente );

})

module.exports = router;

El primer cambio que notaremos es que ahora estamos importando el modelo de Pendiente desde el archivo *pendiente.model.js:*

*//Importamos el modelo de Pendiente.*

**const** Pendiente = require('./pendiente.model');

Esto nos permitirá acceder a todos los métodos/funciones que declaramos en *pendiente.model.js.*

El siguiente cambio es cómo están definidas las rutas. La primera de ellas es la *consulta general:*

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

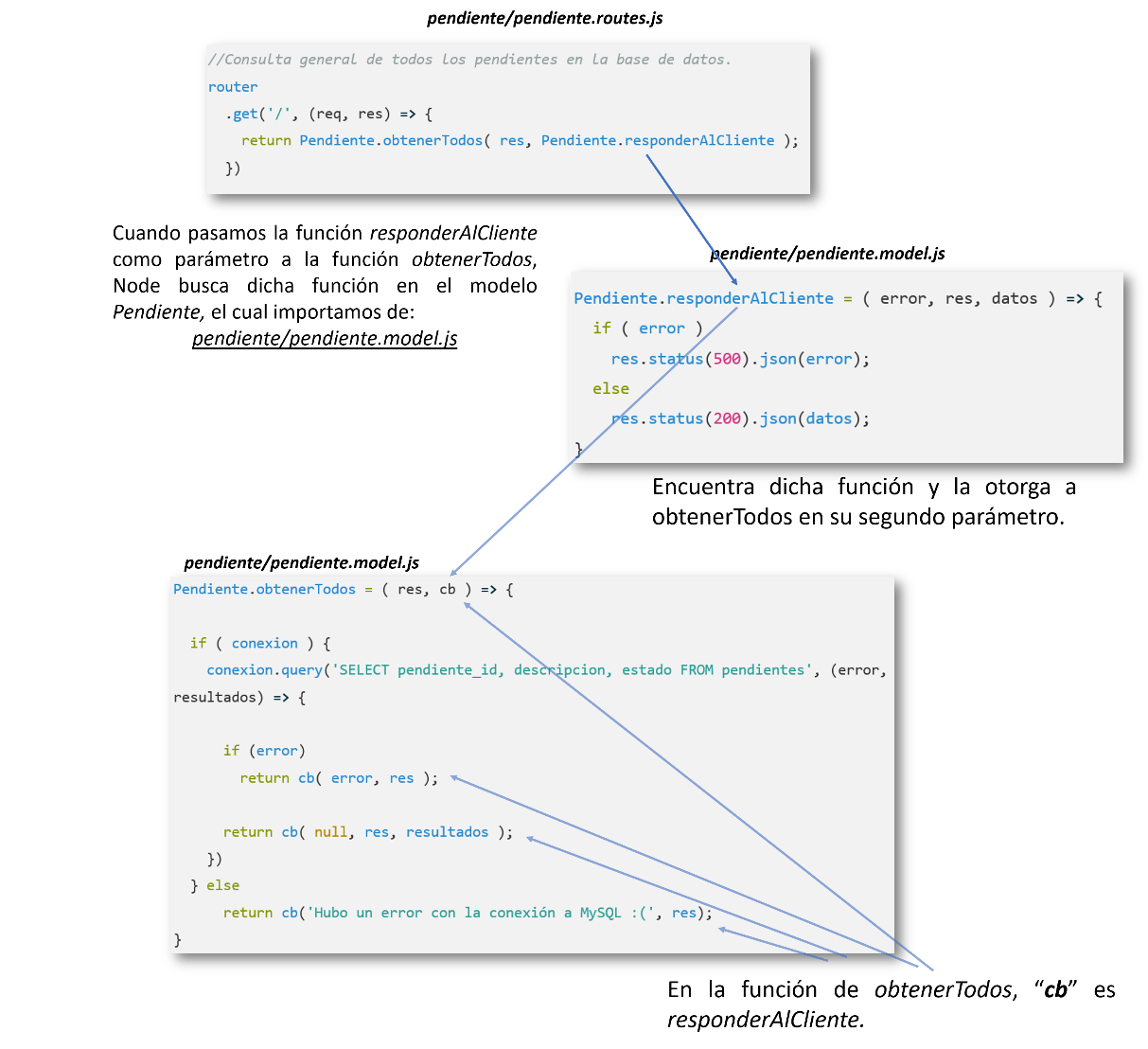
return Pendiente.obtenerTodos( res, Pendiente.responderAlCliente );

})

Como podemos ver, ahora lo único que tenemos que hacer es llamar al método/función obtenerTodos que se encuentra en Pendiente, pasarle como primer parámetro el objeto de ***res*** para poder responderle al cliente, y como segundo parámetro, pasarle el método/función responderAlCliente de Pendiente.

Si miramos con atención, podremos ver que el parámetro llamado “cb” de Pendiente.obtenerTodos = ( res, cb ) **=>** {...}) en el archivo de *pendiente.model.js*, en realidad es la función Pendiente.responderAlCliente. Esta manera de pasar *callbacks* (es decir, funciones que serán llamadas en el futuro) se utiliza muchísimo en aplicaciones de Node.js.

Si te parece confuso, analiza la siguiente imagen:



La siguiente ruta sigue el mismo patrón que la anterior:

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

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

return Pendiente.contador( res, Pendiente.responderAlCliente );

})

Llamamos al método/función contador de Pendiente y le pasamos como parámetro el objeto ***res*** y la función Pendiente.responderAlCliente.

Ocurre lo mismo en la siguiente ruta:

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

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

return Pendiente.siguienteId( res, Pendiente.responderAlCliente );

})

Sólo llamamos a Pendiente.siguienteId y le pasamos los parámetros correspondientes.

En la consulta individual:

*//Consultar un pendiente por id.*

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

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

return Pendiente.encontrarPorId( idPendiente, res, Pendiente.responderAlCliente );

})

Obtenemos el ID de la ruta con req.params.idPendiente y lo pasamos como primer parámetro a Pendiente.encontrarPorId. Después pasamos los parámetros faltantes.

En el registro de un nuevo pendiente:

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

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

**const** nuevoPendiente = {

pendiente\_id: null,

descripcion: req.body.descripcion,

estado: req.body.estado

}

return Pendiente.insertar( nuevoPendiente, res, Pendiente.responderAlCliente );

})

Construimos un nuevoPendiente como lo hemos estado haciendo en los anteriores ejercicios, y después se lo pasamos a Pendiente.insertar como primer parámetro. Enseguida le pasamos res y Pendiente.responderAlCliente como los parámetros faltantes.

En la modificación de un pendiente existente:

*//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

}

return Pendiente.actualizar( pendientePorEditar, res, Pendiente.responderAlCliente);

})

Contruimos pendientePorEditar y se lo pasamos como primer parámetro a Pendiente.actualizar. Después le pasamos res y Pendiente.responderAlCliente como los parámetros restantes.

Finalmente, en la eliminación de un pendiente:

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

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

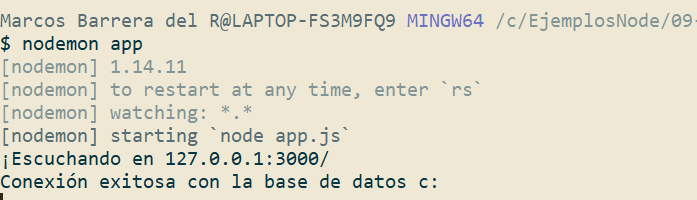
**const** idPendiente = req.params.idPendiente;

return Pendiente.eliminar( idPendiente, res, Pendiente.responderAlCliente);

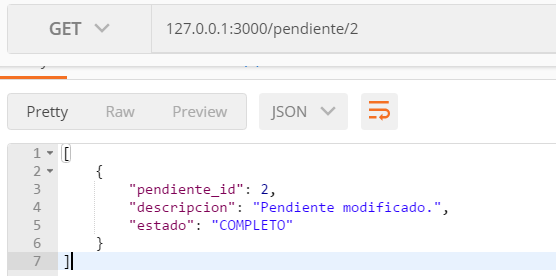
})

Obtenemos el ID desde la URL y se lo pasamos como parámetro a Pendiente.eliminar junto con res y Pendiente.responderAlCliente.

Y para comprobar que sigue funcionando el servidor correctamente, lo ejecutaremos con ***nodemon app***:



Y probaremos el *endpoint* de consulta individual:



Como podemos ver, todo sigue funcionando perfectamente.

Con la implementación de “*Modelos*”podemos mejor drásticamente la facilidad para mantener nuestro código libre de *bugs* y es más manejable.