# Node JS et base de données

## Introduction

Dans le laboratoire précédent, nous avons stocké les données dans des variables. Dans un souci de persistance évident, cette approche est à éviter pour une application en production. Vous devrez donc rapidement vous tourner vers une base de données. Nous allons voir ici comment interagir avec elle.

## Utilisation d’une base de données

Pour illustrer une utilisation de la base de données, nous allons utiliser PostgreSQL. Ce que nous allons voir peut être appliqué à n’importe quelle base de données : MySQL, MariaDB, Cassandra, Mongo, SQLite, etc. Pour commencer, on va préparer la base de données. Placer le fichier Vagrant dans votre projet et dans un terminal utilisez la commande : « vagrant up ». Quand la base de données sera prête, vous pourrez vous y connecter en utilisant les informations suivantes :

* Utilisateur : john
* Mot de passe : password
* Adresse : localhost (ou 127.0.0.1)
* Port : 3000
* Base de données : exercices

Maintenant, on va installer le driver (sous forme de module) qui permet de faire des requêtes facilement à la base de données. Pour cela, on utilise la commande suivante : « npm install -s pg ».

Première chose à faire : créer un fichier « database.js » dans modele. Ce fichier permettra d’avoir un « pool » de connexions. L’avantage d’utiliser un tel système est qu’une connexion peut être réutilisée au lieu d’en recréer une nouvelle. Dans ce fichier, vous ajoutez ce code :

const pg = require("pg");

const Pool = pg.Pool;

const pool = new Pool({

user: 'john',

host: 'localhost',

database: 'exercices',

password: 'password',

port: 3000,

});

module.exports = pool;

Avec cela, la pool est prête et vous pouvez l’utiliser dans votre application. Nous allons peupler la base de données grâce à cela. Créez un dossier appelé « scripts » et dedans créez deux autres dossiers : « JS » et « SQL ». Ensuite, créez un fichier « ./scripts/SQL/createDB.sql » et collez le code suivant :

DROP TABLE IF EXISTS produit CASCADE;

CREATE TABLE produit (

id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,

nom varchar,

prix float

);

INSERT INTO produit (nom, prix) VALUES

('Playstation 4', 400),

('Xbox One', 399.99),

('Nintendo Switch', 349.99);

Maintenant, on va créer le script qui va lire les querys et les faire exécuter. Créez le fichier « ./scripts/JS/initDB.js » et ajoutez le code suivant :

const pool = require("../../modele/database");

const fs = require("fs");

const path = require("path");

async function initDB(){

const client = await pool.connect();

try{

const query = fs.readFileSync(path.join(\_\_dirname, "../SQL/createDB.SQL"), "utf-8");

await client.query(query);

} catch (e) {

console.error(e);

} finally {

client.release();

await pool.end();

}

}

initDB().then(() => console.log("done"));

Ce que fait le code est assez simple. Il y a une fonction asynchrone qui :

* Demande un client à la pool
* Ensuite il y a lecture du fichier SQL qu’on a créé et on stocke le contenu dans une variable
* Enfin on demande au client d’exécuter la query
* Si on a une erreur : on affiche l’erreur
* Dans tous les cas, on libère le client et on termine la pool
* Quand la fonction asynchrone est finie, on affiche « done » dans la console.

Maintenant que le script est prêt, on va le rendre facile d’accès en l’ajoutant au fichier package.json. A l’intérieur de ce fichier, ajouter la ligne suivante dans la partie « scripts » : "initDB": "node scripts/JS/initDB.js". Si on fait un « npm run initDB » le script crée la table et y ajoute les données.

Ensuite, on va devoir mettre à jour les parties suivantes : les modèles (on va devoir utiliser un client), les contrôleurs (il faudra créer un client et le passer aux différents modèles) et les routes (en effet, les fonctions vont devenir asychrones).

Créez un fichier « ./modele/produitDB.js » et mettez le code suivant :

module.exports.getProduit = async (id, client) => {

return await client.query("SELECT \* FROM produit WHERE id = $1", [id]);

}

Vous devrez essayer de transformer les autres 3 méthodes manquantes (« postProduit », « updatePrix » et « deleteProduit ») en vous inspirant du code donné. N’oubliez pas d’aller voir les paramètres des fonctions dans le fichier « ./modele/produit.js » et d’utiliser les requêtes paramétrisées comme dans l’exemple.

Quand cela est fait, vous pouvez créer le fichier « ./controleur/produitDB.js » et vous pouvez ajouter le code suivant dedans :

module.exports.getProduit = async (req, res) => {

const client = await pool.connect();

const idTexte = req.params.id; //attention ! Il s'agit de texte !

const id = parseInt(idTexte);

try{

if(isNaN(id)){

res.sendStatus(400);

} else {

const {rows: produits} = await ProduitModele.getProduit(id, client);

const produit = produits[0];

if(produit !== undefined){

res.json(produit);

} else {

res.sendStatus(404);

}

}

} catch (error){

console.error(error);

res.sendStatus(500);

} finally {

client.release();

}

}

Maintenant, la fonction du contrôleur est devenue asynchrone, ce qui permet d’utiliser le mot clé « await ». Avec cet exemple et en vous inspirant du fichier « ./controleur/produit.js », vous êtes capables de faire les trois fonctions manquantes : « postProduit », « updateProduit », « deleteProduit ».

Enfin, dernière étape : mettre à jour les routes (dans « ./route/produit.js ») pour utiliser le nouveau contrôleur :

//const ProduitControleur = require("../controleur/produit");

const ProduitControleur = require("../controleur/produitDB");

//const router = require("express").Router();

const Router = require("express-promise-router");

const router = new Router;

Pour que cela fonctionne, vous avez besoin d’installer le nouveau paquet avec cette commande : « npm install –save express-promise-router ».

Et voilà, vous avez un système qui marche complètement avec une base de données. Votre application n’est pas encore parfaite : il manque un système d’identification par exemple (nous verrons cela plus tard). La suite du laboratoire visera à approfondir l’utilisation de la base de données avec NodeJS.

## Informations sur la connexion à la base de données

Actuellement les informations sont stockées en dur dans l’application. Ceci est une mauvaise approche du point de vue de la sécurité. En effet, si un hacker venait à prendre possession de votre répertoire git, il aurait un accès à votre base de données. Il existe pour cela les variables d’environnement qui permettent de créer des variables pour un processus. Le problème de cette approche est qu’il faut préciser l’ensemble des variables à chaque fois que l’on lancera le processus. Dans une phase de développement, cela peut s’avérer assez pénible. Dans une phase de production, si les serveurs redémarrent vous devrez relancer le processus manuellement. Il existe une solution assez simple à ce problème qui consiste à créer un fichier « .env » à la racine de votre projet et d’y indiquer les informations nécessaires. Les valeurs indiquées seront chargées par un paquet. Pour cela, utilisez la commande : «npm i dotenv» et créez le fichier « .env » à la racine de votre projet. Quand cela sera fait, référez-vous à la documentation du paquet ( <https://www.npmjs.com/package/dotenv> ) pour comprendre comment compléter le fichier et comment utiliser les variables d’environnement. Sachez juste qu’avec la configuration actuelle, vous n'aurez besoin de modifier que le fichier «./modele/database.js».

Ce fichier « .env » ne doit donc pas être mis sur un git étant donné qu’il contient les informations secrètes. Par contre, il est fortement conseillé de préciser quelque part (README.md, par exemple) qu’il faut un fichier « .env » et d’y indiquer quelles variables il faut ajouter.

## Différence entre pool et client

Dans certains frameworks, dont celui utilisé en exemple, on utilise le terme « pool » et « client ». La différence est importante surtout si vous faites plusieurs opérations dans une transaction. Un client est une connexion vers la base de données. Quand vous faites plusieurs opérations avec un même client, c’est équivalent à faire une requête avec plusieurs instructions séparées par un « ; ». C’est également obligatoire si vous désirez récupérer une information pour l’utiliser dans une opération par la suite. Une pool sert à récupérer un client disponible et à le libérer (c’est-à-dire qu’il sera marqué comme libre) pour qu’il puisse être utilisé à nouveau. Il faut donc toujours veiller à libérer le client lorsque l’on a fini.

Le module « pg » permet de demander à une pool d’exécuter une requête. En réalité, il s’agit d’une facilité pour faire en une opération les étapes suivantes :

* Acquisition d’un client
* Exécution de la requête et récupération du résultat
* Libération du client

Il est donc important de bien utiliser un client et non une pool pour effectuer une série d’opérations d’une transaction.

## Accès complexe à la base de données : transaction

Pour simuler un accès complexe, nous allons créer deux tables en plus de celle que nous avions déjà. L’idée est d’avoir une table « client » et « achat » pour représenter les commandes. Dès lors, nous allons avoir besoin des transactions (un client doit avoir fait un achat et un achat doit être fait par un client) pour pouvoir réaliser l’exercice de manière correcte.

D’abord, nous allons améliorer notre fichier SQL pour ajouter ce qui manque. Ajoutez donc ces lignes :

DROP TABLE IF EXISTS client CASCADE;

CREATE TABLE client(

id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,

nom varchar,

prenom varchar,

adresse varchar

);

INSERT INTO client (nom, prenom, adresse) VALUES ('Poirier', 'Tevin', '11, rue du Faubourg National 95150 TAVERNY');

DROP TABLE IF EXISTS achat CASCADE;

CREATE TABLE achat (

id\_produit integer REFERENCES produit(id) DEFERRABLE INITIALLY IMMEDIATE,

id\_client integer REFERENCES client(id) DEFERRABLE INITIALLY IMMEDIATE,

quantite integer,

"date" date,

PRIMARY KEY(id\_client, id\_produit, "date")

);

Après cela, utilisez la commande « npm run initDB » et maintenant votre base de données contient les tables supplémentaires.

Etant donné que les nouveaux modèles, contrôleurs et routeurs n’incluront pas toutes les méthodes d’un CRUD (le but étant seulement de montrer la gestion d’une transaction) et qu’ils se calquent fortement sur les fichiers précédents, vous n’êtes pas obligés de les faire vous-mêmes (cependant, c’est un bon exercice). Seule la partie de la gestion de la transaction sera à faire. Voici les fichiers et leur contenu :

* « ./controleur/achatDB.js »

const pool = require('../modele/database');

const ClientModele = require("../modele/clientDB");

const AchatModele = require("../modele/achatDB");

module.exports.insertAchat = async (req, res) => {

const client = await pool.connect();

const {idProduit, client:clientObj, quantite} = req.body;

//TO DO

}

* « ./modele/achatDB.js »

module.exports.insertAchat = async (client, id\_client, id\_produit, quantite, date = new Date().toISOString()) => {

return await client.query(`

INSERT INTO achat(id\_client, id\_produit, quantite, date) VALUES

($1, $2, $3, $4)`, [id\_client, id\_produit, quantite, date]

);

}

* « ./modele/clientDB.js »

module.exports.createClient = async (client, nom, prenom, adresse) => {

return await client.query(`

INSERT INTO client(nom, prenom, adresse)

VALUES ($1, $2, $3)`, [nom, prenom, adresse]

);

}

module.exports.clientExist = async (client, idClient) => {

const {rows} = await client.query(

"SELECT count(id) AS nbr FROM client WHERE id = $1",

[idClient]

);

return rows[0].nbr > 0;

}

* « ./route/achat.js »

const AchatControleur = require("../controleur/achatDB");

const Router = require("express-promise-router");

const router = new Router;

router.post('/', AchatControleur.insertAchat);

module.exports = router;

N’oubliez pas d’inclure le nouveau routeur dans le gestionnaire de route « ./route/index.js ».

Essayez de vous concentrer sur les cas suivants :

* Faire l’opération d’insérer l’achat dans une transaction (même si l’utilité est discutable en regardant la structure de la DB)
* Gérer le cas où le client proposé n’existe pas et envoyez une erreur explicative. La documentation d’Express couvre ce cas : <https://expressjs.com/en/4x/api.html#res.send>
* Utilisez un try/catch pour capturer des éventuelles erreurs
* N’oubliez pas de Rollback/Commit et d’envoyer une réponse adéquate

Quand vous aurez fini, vous pouvez consulter la solution à la page suivante. Il existe plusieurs implémentations possibles, le but étant d’avoir une solution pour les étudiants qui bloquent ou qui souhaitent voir une autre approche possible.

module.exports.insertAchat = async (req, res) => {

const client = await pool.connect();

const {idProduit, client:clientObj, quantite} = req.body;

try {

await client.query("BEGIN;");

const clientExist = await ClientModele.clientExist(client, clientObj.id);

if (clientExist === true) {

await AchatModele.insertAchat(

client,

clientObj.id,

idProduit,

quantite

);

await client.query("COMMIT");

res.sendStatus(201);

} else {

await client.query("ROLLBACK");

res.status(404).json({error: "l'id du client n'existe pas"});

}

} catch (e) {

await client.query("ROLLBACK;");

console.error(e);

res.sendStatus(500);

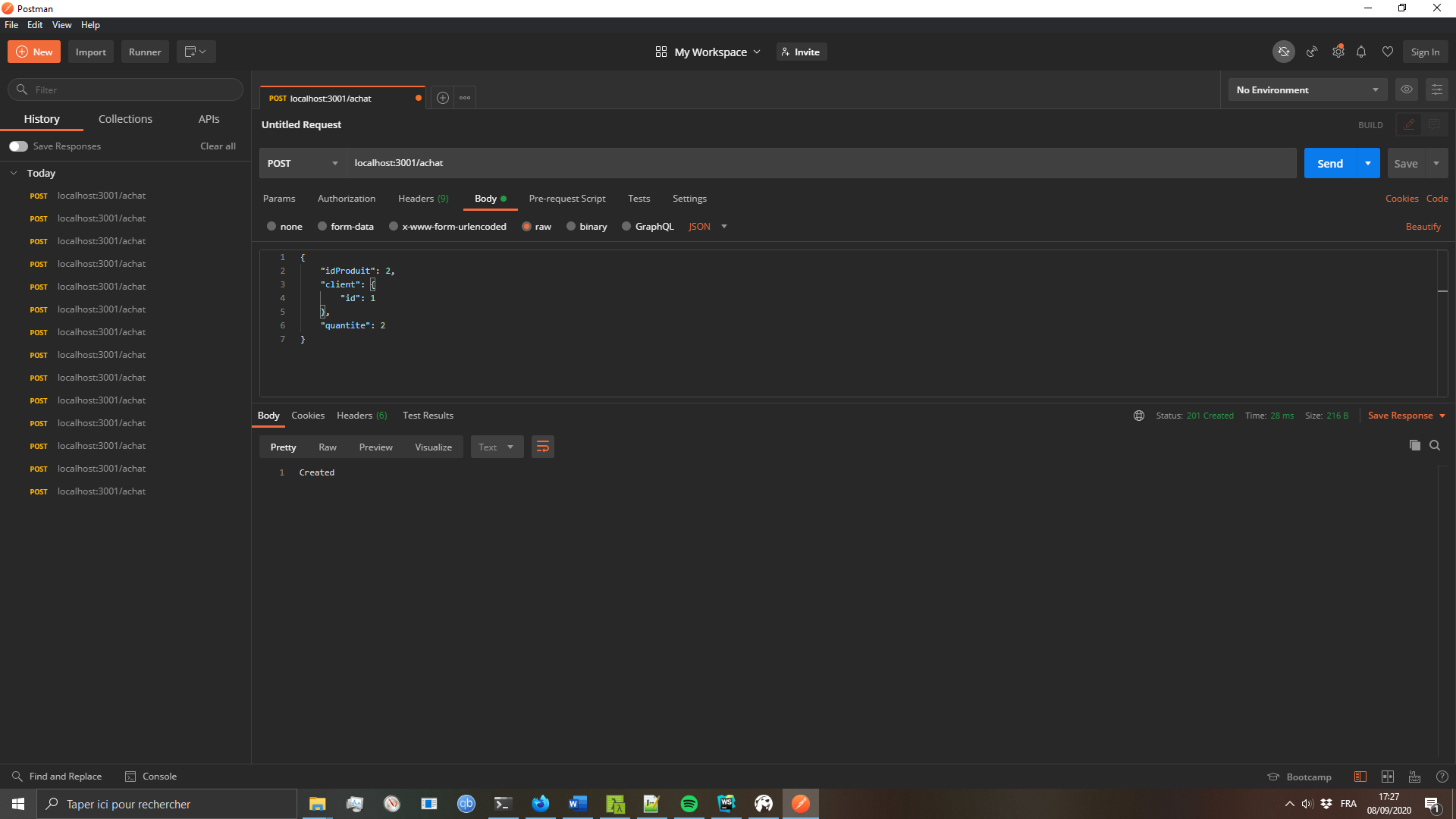
} finally {

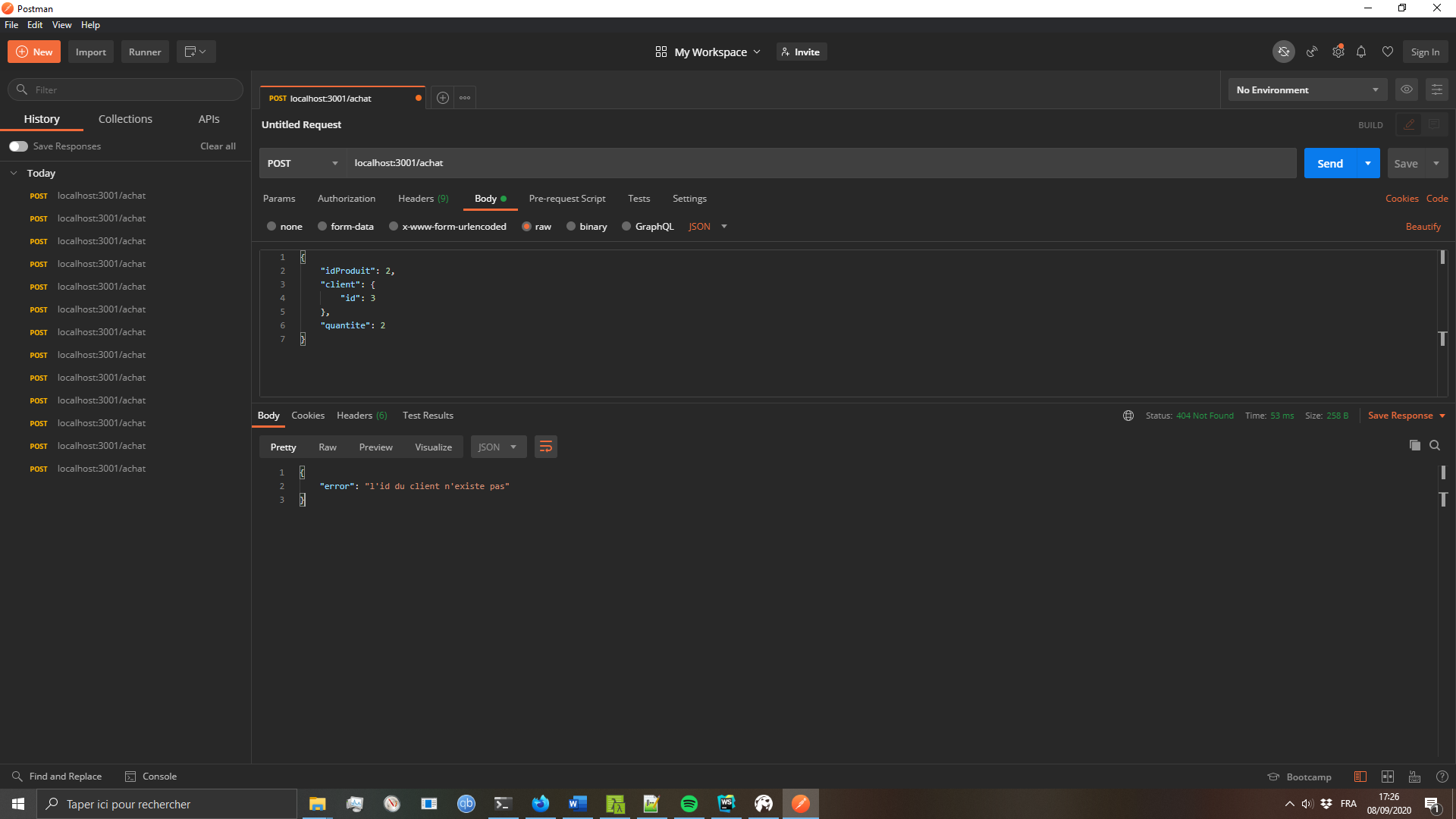
client.release();

}

}

Et voici le résultat dans postman :





## Encapsuler le client de la base de données

Cette partie du cours a pour but d’aller un peu plus loin dans l’analyse du code proposé plus haut.

Si on regarde de plus près le code, nous voyons rapidement qu’il y a une forte dépense à la base de données employée. Il en va de même avec le client « pg ». Si l’un des deux changent, il faudra faire un certain nombre de modifications. Imaginez que la transaction ne commence pas par « BEGIN ; » mais par « START ;». Il faudra donc modifier tous les endroits où nous effectuons des transactions. Pour éviter cela, on aura tendance à encapsuler le client de base de données dans un objet qu’on aura créé.

Dans l’exemple donné précédent, on aura un appel de méthode qui serait « monClient.beginTransaction() » qui contiendrait toute la logique pour commencer une transaction. Si vous devez modifier le « BEGIN ; » en « START ; » il suffira juste de changer le corps de la méthode et non les multiples appels à cette fonction. Si on continue avec cette approche, il est possible de devenir presque indépendant du client ou de la base de données.