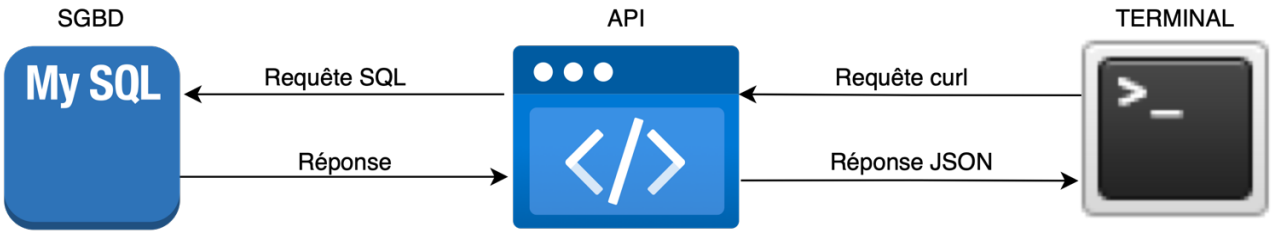
# CORRIGÉ CODAGE D'UNE API REST




## Création de la base de donnée Mysql

1. Créer une base de donnée que l'on nommera : students_db à partir de terminal.

2. Créer une table nommée students ( id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(255) NOT NULL, email VARCHAR(255) NOT NULL UNIQUE, age INT NOT NULL) à partir du terminal.

3. Insérer quelques données pour tester :

('Alice', 'alice@example.com', 20), ('Bob', 'bob@example.com', 22).

## Création du projet :

1. Créer un répertoire api.
2. Créer dans ce répertoire une projet nodejs : `npm init -y`
3. Installer les modules express mysql2 dotenv.

## Création de l'API REST :

### Configuration de la connexion à MySQL

1. Créer un fichier .env pour stocker les informations de connexion :

```
DB_HOST=localhost
DB_USER=ciel
DB_PASSWORD=ciel
DB_NAME=students_db
```

2. Créer un fichier db.js pour gérer la connexion :

```
const mysql = require('mysql2');
const dotenv = require('dotenv');

dotenv.config();

const pool = mysql.createPool({
  host: process.env.DB_HOST,
  user: process.env.DB_USER,
```

```
  password: process.env.DB_PASSWORD,
  database: process.env.DB_NAME,
});

module.exports = pool.promise();
```

***Explications du code :***

**Création d'un pool de connexions**

*dotenv.config(); // Charge les variables d'environnement avant de les utiliser*

*const pool = mysql.createPool({*
*  host: process.env.DB_HOST,*
*  user: process.env.DB_USER,*
*  password: process.env.DB_PASSWORD,*
*  database: process.env.DB_NAME,*
*});*

***Qu'est-ce qu'un pool de connexions ?***

*Un **pool de connexions** est un ensemble de connexions pré-créées à la base de données.*

- *Avantage : Cela améliore les performances car il n'est pas nécessaire d'ouvrir et de fermer une connexion à chaque requête. Les connexions sont réutilisées.*

***Les options utilisées :***

- ***host** : L'adresse de votre serveur MySQL.*
- ***user** : Le nom d'utilisateur MySQL.*
- ***password** : Le mot de passe associé à l'utilisateur.*
- ***database** : Le nom de la base de données à utiliser.*

***Pourquoi process.env ?***

- ***process.env** est utilisé pour lire les variables d'environnement définies dans le fichier .env.*
    - *Avantage : Les informations sensibles ne sont pas directement dans le code source. Cela permet également de changer les configurations sans modifier le code.*

***Utilisation de pool.promise()***

*En appelant .promise(), on dit à mysql2 de fonctionner en mode **Promise**. Cela nous permettrai de faire par exemple :*

*const [rows] = await pool.query('SELECT * FROM table_name');*

***Exportation du pool***

*module.exports = pool.promise();*

*module.exports : Permet de rendre le pool disponible dans d'autres fichiers.*

**Création de l'API :**

1. Créer un fichier server.js pour gérer la connexion :

```
const db = require('./db');

async function getStudents() {
  const [rows] = await db.query('SELECT * FROM students');
  console.log(rows);
}

getStudents();
```

***Explications du code :***

- ***async function***
  - *L'ajout de async transforme getStudents en une fonction asynchrone.*
  - *Cela permet d'utiliser await à l'intérieur de la fonction pour gérer des Promises de manière plus lisible.*

- ***db.query : Appelle une méthode de requête MySQL depuis le pool de connexions.***
  - *db.query('SELECT * FROM students') envoie une requête SQL pour récupérer tous les enregistrements de la table students.*
  - *Cette méthode renvoie une Promise contenant deux éléments :*
    1. *rows : Les résultats de la requête sous forme d'un tableau d'objets.*
    2. *fields (non utilisé ici) : Les métadonnées des colonnes de la requête.*

- ***await attend que la Promise retournée par db.query soit résolue.***
  - *Cela permet d'obtenir directement les résultats sous forme de tableau (rows), sans utiliser .then().*

- ***Destructuration [rows]***
  - *La syntaxe [rows] extrait directement la première valeur renvoyée par la Promise (les lignes de la requête SQL).*

2. Modifier le fichier server.js :

```
const express = require('express');
const dotenv = require('dotenv');
const db = require('./db');

const app = express();
app.use(express.json());
```

```
const PORT = process.env.PORT || 3000;

// Routes
app.get('/', (req, res) => {
  res.send('API REST Node.js + MySQL');
});

// Démarrer le serveur
app.listen(PORT, () => {
  console.log(`Serveur démarré sur http://localhost:${PORT}`);
});
```

3. Tester maintenant l'api depuis un navigateur : http://localhost:3000

4. Ajouter le code suivant dans server.js :

```
app.get('/students', async (req, res) => {
try {
const [rows] = await db.query('SELECT * FROM students');
res.json(rows);
} catch (err) { res.status(500).json({ error: err.message }); }
});
```

5. Expliquer le code précédent.

Déclaration de la route GET :

```
app.get('/students', async (req, res) => { ... });
```

- **app.get()** :
  - Définit une route HTTP de type **GET** dans l'application **Express**.

**'/students'** :

- Spécifie le chemin ou endpoint auquel cette route répondra.
- Par exemple, une requête HTTP GET envoyée à http://localhost:3000/students déclenchera cette fonction.

**async (req, res)** :

- Déclare une fonction asynchrone. Le mot-clé async permet d'utiliser await à l'intérieur de la fonction.
- **req** : L'objet représentant la requête HTTP (paramètres, en-têtes, etc.).
- **res** : L'objet utilisé pour envoyer une réponse HTTP au client.

Bloc try : Récupérer les données

```
try {
  const [rows] = await db.query('SELECT * FROM students');
  res.json(rows);
}
```

**try** :

- Tout code susceptible de générer une erreur est placé dans ce bloc. Cela permet de gérer les erreurs proprement via un bloc catch.

**await db.query('SELECT * FROM students')** :

- **db.query()** :
  - Exécute une requête SQL sur la base de données.
  - Ici, la requête SQL est :

```
SELECT * FROM students
```

Elle récupère toutes les lignes de la table students.

- **await** :
  - Attend que la requête soit exécutée et que la réponse de la base de données soit disponible avant de continuer.
- [**rows**] :
  - La méthode db.query() retourne une Promise qui, une fois résolue, contient :
    1. **rows** : Les résultats de la requête SQL sous forme d'un tableau d'objets.
    2. **fields** : Métadonnées des colonnes SQL (non utilisé ici).
  - **Destructuration** [**rows**] :
    - Extraire directement les résultats de la requête.

Exemple de contenu de rows :

```
[
  { id: 1, name: 'Alice', email: 'alice@example.com', age: 20 },
  { id: 2, name: 'Bob', email: 'bob@example.com', age: 22 }
]
```

Bloc catch : Gestion des erreurs

```
catch (err) {
  res.status(500).json({ error: err.message });
}
```

**catch** :

- Ce bloc capture les erreurs qui peuvent survenir dans le bloc try.

**err** :

- L'objet représentant l'erreur.

**res.status(500)** :

- Définit le code de statut HTTP comme **500 (Internal Server Error)** pour indiquer qu'une erreur serveur s'est produite.

**res.json({ error: err.message })** :

- Envoie une réponse JSON contenant un message d'erreur.
- Exemple de réponse en cas d'erreur :

```
{
  "error": "Cannot connect to database"
}
```

6. Indiquer la signification du code 500.

Le code 500 indique une erreur interne du serveur.

7. Tester le code à l'aide d'une requête curl :

```
curl –X GET http://localhost:3000/students
```

8. Ajouter le code suivant au fichier server.js :

```
app.post('/students', async (req, res) => {
  const { name, email, age } = req.body;
  try {
    const sql = `INSERT INTO students (name, email, age) VALUES
('${name}', '${email}', ${age})`;
    const [result] = await db.query(sql);
    res.status(201).json({ id: result.insertId, name, email, age });
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});
```

9. Expliquer le code précédent.

Extraction des données du corps de la requête :

```
const { name, email, age } = req.body;
```

**req.body** :

- Contient les données envoyées par le client dans le corps de la requête HTTP.
- Pour que req.body fonctionne, un middleware comme express.json() doit être configuré dans l'application :

```
app.use(express.json());
```

**Destructuration** :

- Les propriétés name, email, et age sont extraites directement du corps de la requête.
- Exemple de contenu possible de req.body :

```
{
  "name": "Alice",
  "email": "alice@example.com",
  "age": 20
}
```

Bloc try : Insertion dans la base de données

```
try {
  const sql = `INSERT INTO students (name, email, age) VALUES
('${name}', '${email}', ${age})`;
  const [result] = await db.query(sql);
  res.status(201).json({ id: result.insertId, name, email, age });
}
```

**Construction de la requête SQL**

```
const sql = `INSERT INTO students (name, email, age) VALUES
('${name}', '${email}', ${age})`;
```

- La requête SQL est construite comme une chaîne de caractères.
- Les données de name, email, et age sont directement insérées dans la requête :

```
INSERT INTO students (name, email, age) VALUES ('Alice',
'alice@example.com', 20)
```

**Problème potentiel : Injection SQL**

- Cette méthode est vulnérable aux attaques par injection SQL si name, email, ou age contiennent des données malveillantes.
- Par exemple, un attaquant pourrait envoyer :

```
{
  "name": "Alice'; DROP TABLE students; --",
  "email": "alice@example.com",
  "age": 20
}
```

Cela générerait une requête SQL dangereuse :

```
INSERT INTO students (name, email, age) VALUES ('Alice'; DROP TABLE students; --', 'alice@example.com', 20)
```

**Envoi d'une réponse avec succès**

res.status(201).json({ id: result.insertId, name, email, age });

**res.status(201)** :

- Définit le code HTTP comme **201 Created**, indiquant qu'une ressource a été créée avec succès.

**res.json()** :

- Envoie une réponse JSON au client contenant :
  - L'ID de la ligne insérée (result.insertId).
  - Les données fournies (name, email, age).

Exemple de réponse JSON envoyée au client :

```
{
  "id": 1,
  "name": "Alice",
  "email": "alice@example.com",
  "age": 20
}
```

10. Indiquer la signification du code 201.

- le code HTTP comme **201 Created**, indique qu'une ressource a été créée avec succès.

1. Tester le code à l'aide d'une requête curl :

```
curl -X POST http://localhost:3000/students \
-H "Content-Type: application/json" \
-d '{"name": "Alice", "email": "alice@example.com", "age": 20}'
```

2. Ajouter le code suivant au fichier server.js :

```
app.get('/students/:id', async (req, res) => {
  try {
    // Construire la requête SQL directement
    const sql = `SELECT * FROM students WHERE id =${req.params.id}`;
```

```
    // Exécuter la requête
    const [rows] = await db.query(sql);

    // Vérifier si l'étudiant existe
    if (rows.length === 0) {
      return res.status(404).json({ error: 'Student not found' });
    }

    // Retourner les données de l'étudiant
    res.json(rows[0]);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});
```

3. Expliquer le code précédent.

**'/students/:id'** :

- Spécifie un chemin avec un paramètre dynamique (:id), qui correspond à l'ID de l'étudiant à récupérer.
- Par exemple, une requête à **http://localhost:3000/students/1** attribue la valeur **1** à req.params.id.

**async (req, res)** :

- Déclare une fonction asynchrone, ce qui permet d'utiliser await pour les appels asynchrones.

  **req** : Contient les informations de la requête, dont le paramètre id.
  **res** : Utilisé pour envoyer une réponse HTTP au client.

  Construction de la requête SQL
  `const sql = \`SELECT * FROM students WHERE id =${req.params.id}\`;`
  **req.params.id** :
- Contient la valeur du paramètre dynamique :id dans l'URL.

  La requête SQL est construite pour sélectionner l'étudiant correspondant à cet ID :
  `SELECT * FROM students WHERE id = 1`
  **Problème potentiel : Injection SQL**
- Si req.params.id contient une valeur malveillante comme 1 OR 1=1, la requête générée devient dangereuse :
  `SELECT * FROM students WHERE id = 1 OR 1=1`
  Cela pourrait retourner toutes les lignes de la table students, compromettant la sécurité des données.

4. Indiquer et tester la requête curl à saisir pour sélectionner l'étudiant 2.

```
curl -X GET http://localhost:3000/students/2
```

5. Ajouter le code suivant au fichier server.js :

```
app.put('/students/:id', async (req, res) => {
  const { name, email, age } = req.body;
```

```
  const id = req.params.id;

  try {
    // Construire la requête SQL directement
    const sql = `UPDATE students SET name = '${name}', email =
'${email}', age = ${age} WHERE id = ${id}`;

    // Exécuter la requête
    const [result] = await db.query(sql);

    // Vérifier si un étudiant a été mis à jour
    if (result.affectedRows === 0) {
      return res.status(404).json({ error: 'Student not found' });
    }

    // Répondre avec succès
    res.json({ message: 'Student updated' });
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});
```

6. Expliquer le code précédent.

**app.put()** :

- Définit une route HTTP de type **PUT**.
- La méthode PUT est utilisée pour mettre à jour des ressources existantes.

'**/students/:id**' :

- Spécifie un chemin avec un paramètre dynamique :id.
- Cela permet de capturer un ID dans l'URL. Par exemple, une requête à **http://localhost:3000/students/1**attribue la valeur **1** à req.params.id.

**async (req, res)** :

- Déclare une fonction asynchrone qui permet d'utiliser await pour les opérations asynchrones.

**Extraction des données**

```
const { name, email, age } = req.body;
const id = req.params.id;
```

**req.body** :

- Contient les données envoyées dans le corps de la requête HTTP, comme le nom, l'email et l'âge de l'étudiant.
- Nécessite que **express.json()** soit configuré dans votre application pour analyser le corps des requêtes JSON.

**Destructuration** :

- Les propriétés name, email, et age sont extraites directement du corps de la requête.
- Exemple de contenu de req.body :

```
{
  "name": "Alice Updated",
  "email": "alice.updated@example.com",
```

```
  "age": 21
}
```

**req.params.id** :

- Contient l'ID dynamique de l'étudiant à mettre à jour, transmis dans l'URL.

**Construction de la requête SQL**

```
const sql = `UPDATE students SET name = '${name}', email =
'${email}', age = ${age} WHERE id = ${id}`;
```

La requête SQL est construite en insérant directement les valeurs dans la chaîne SQL : UPDATE students SET name = 'Alice Updated', email = 'alice.updated@example.com', age = 21 WHERE id = 1;

**Problème potentiel : Injection SQL**

- Ce code est vulnérable aux attaques SQL si name, email, age, ou id contiennent des données malveillantes. Par exemple, si un utilisateur envoie name = "Alice', DROP TABLE students; --", cela pourrait entraîner des dommages à la base de données.

**Vérification des résultats**

```
if (result.affectedRows === 0) {
  return res.status(404).json({ error: 'Student not found' });
}
```

**result.affectedRows** :

- Indique le nombre de lignes affectées par la requête SQL.
- Si affectedRows === 0, cela signifie que l'ID fourni (req.params.id) ne correspond à aucun étudiant dans la table.

**Réponse avec une erreur 404 :**

- Si aucun étudiant n'est trouvé, une réponse avec le statut **404 Not Found** est renvoyée :

```
{
  "error": "Student not found"
}
```

7. Indiquer et tester la requête curl à saisir pour modifier l'âge de l'étudiant1.

```
curl -X PUT http://localhost:3000/students/1 \
-H "Content-Type: application/json" \
-d '{"name": "Alice Updated", "email": "alice.updated@example.com",
"age": 21}'
```

8. Proposer le code pour effacer un enregistrement.

```
app.delete('/students/:id', async (req, res) => {
try {
    const [result] = await db.query('DELETE FROM students WHERE id =
?', [req.params.id]);
if (result.affectedRows === 0) {
```

```
return res.status(404).json({ error: 'Student not found' });
}
res.json({ message: 'Student deleted' });
} catch (err) {
res.status(500).json({ error: err.message });
}
});
```

9. Indiquer et tester la requête curl à saisir pour modifier l'âge de l'étudiant2.

```
curl -X DELETE http://localhost:3000/students/1
```