Projet :
Création d'une BD
pour une API RESTful
en Express.js

Sommaire

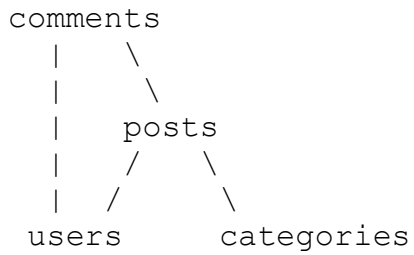
Table des matières

Création d'une BD pour une API RESTful en Express.js

Cette BD permettra de gérer :
✅ **Utilisateurs** (création de compte, mise à jour, suppression)
✅ **Articles (posts)** rédigés par des utilisateurs
✅ **Commentaires** sur les articles
✅ **Catégories** d'articles

☐Conception de la Base de Données (Modèle SQL)

Schéma de la BD :



On crée les **tables essentielles** avec leurs relations :

```
CREATE DATABASE blog_api;
USE blog_api;

-- Table des utilisateurs
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Table des catégories d'articles
CREATE TABLE categories (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL UNIQUE
);

-- Table des articles (posts)
CREATE TABLE posts (
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(255) NOT NULL,
    content TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    user_id INT,
    category_id INT,
    FOREIGN KEY (user_id) REFERENCES users(id),
    FOREIGN KEY (category_id) REFERENCES categories(id));

-- Table des commentaires
CREATE TABLE comments (
    id INT AUTO_INCREMENT PRIMARY KEY,
    post_id INT,
```

```
    user_id INT,
    content TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    primary KEY(post_id, user_id),
    FOREIGN KEY (post_id) REFERENCES posts(id),
    FOREIGN KEY (user_id) REFERENCES users(id)
);
```

2️⃣ API RESTful : Routes pertinentes

Voici les opérations **CRUD** que l'API devra proposer :

📌 Gestion des Utilisateurs

- `GET /users` → Liste des utilisateurs
- `GET /users/:id` → Récupérer un utilisateur
- `POST /users` → Ajouter un utilisateur
- `PUT /users/:id` → Mettre à jour un utilisateur
- `DELETE /users/:id` → Supprimer un utilisateur

📌 Gestion des Articles

- `GET /posts` → Liste des articles
- `GET /posts/:id` → Détails d'un article
- `POST /posts` → Ajouter un article
- `PUT /posts/:id` → Modifier un article
- `DELETE /posts/:id` → Supprimer un article

Endpoints avancés :

- `GET /posts?category=tech` → Filtrer les articles par catégorie
- `GET /users/:id/posts` → Récupérer les articles d'un utilisateur

📌 Gestion des Commentaires

- `GET /posts/:id/comments` → Liste des commentaires d'un article
- `POST /posts/:id/comments` → Ajouter un commentaire
- `DELETE /comments/:id` → Supprimer un commentaire

3️⃣ Exemple de Données (DUMP SQL)

On peut insérer des **données de test** :

```
-- Insérer des utilisateurs
INSERT INTO users (name, email, password) VALUES
('Alice', 'alice@example.com', 'hashedpassword1'),
('Bob', 'bob@example.com', 'hashedpassword2');

-- Insérer des catégories
INSERT INTO categories (name) VALUES
('Tech'),
('Lifestyle'),
('Science');

-- Insérer des articles
INSERT INTO posts (user_id, category_id, title, content) VALUES
(1, 1, 'Les nouveautés en IA', 'Blabla sur l\'intelligence
artificielle...'),
(2, 2, '10 astuces pour mieux dormir', 'Blabla sur le sommeil...');

-- Insérer des commentaires
INSERT INTO comments (post_id, user_id, content) VALUES
(1, 2, 'Super article, merci !'),
(2, 1, 'Intéressant, merci pour les conseils.');
```

4️⃣ Résumé des Requêtes SQL pour API

Consultation (SELECT)

```
SELECT * FROM users; -- Liste des utilisateurs
SELECT * FROM posts; -- Liste des articles
SELECT * FROM posts WHERE category_id = 1; -- Articles d'une catégorie
SELECT * FROM comments WHERE post_id = 1; -- Commentaires d’un article
```

Ajout (INSERT)

```
INSERT INTO users (name, email, password) VALUES ('Charlie',
'charlie@example.com', 'hashedpassword');
INSERT INTO posts (user_id, category_id, title, content) VALUES (3, 1, 'Un
nouvel article', 'Contenu ici...');
INSERT INTO comments (post_id, user_id, content) VALUES (3, 1, 'Mon avis
sur cet article...');
```

Mise à jour (UPDATE)

```
UPDATE users SET name = 'Alice Dupont' WHERE id = 1;
UPDATE posts SET title = 'Titre modifié' WHERE id = 2;
```

Suppression (DELETE)

```
DELETE FROM users WHERE id = 3;
DELETE FROM posts WHERE id = 2;
DELETE FROM comments WHERE id = 1;
```

🎯 Pourquoi cette BD est adaptée pour une API RESTful ?

✅ Relations claires entre utilisateurs, articles et commentaires
✅ Possibilité d'ajouter des filtres (`?category=tech`)
✅ Gestion CRUD simple et efficace
✅ Extensible facilement (ajout d’un système de "likes", etc.)

Création d’une API RESTful en Express.js

On va maintenant créer une **API RESTful en Express.js** pour interagir avec la base de données MySQL que nous avons définie.

📌 Outils utilisés

✅ **Node.js** avec **Express** pour gérer l'API
✅ **MySQL2** pour se connecter à la base de données
✅ **Postman ou cURL** pour tester les endpoints

Le code proposé ci-dessous n’est pas en MVC.

Il propose une solution qu’on écrira en MVC en suivant les principes du cours.

☐Installation des dépendances

Dans le projet Node.js, on installe les modules nécessaires :

```
npm init -y  # Initialise le projet
npm install express mysql2 dotenv body-parser cors
```

- `express` → framework pour créer l'API
- `mysql2` → module pour interagir avec MySQL
- `dotenv` → gérer les variables d’environnement
- `body-parser` → analyser les données JSON envoyées dans les requêtes
- `cors` → permettre les requêtes cross-origin

2️⃣ Configuration de la connexion à MySQL

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

```
DB_HOST=localhost
DB_USER=root
DB_PASSWORD=
DB_NAME=blog_api
```

Ensuite, créer un fichier `db.js` pour gérer la connexion MySQL :

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

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

connection.connect(err => {
    if (err) {
        console.error('❌ Erreur de connexion à MySQL :', err);
        return;
    }
    console.log('✅ Connecté à MySQL');
});

module.exports = connection;
```

3️⃣ Création du serveur Express

Crée un fichier **`server.js`** et configure Express :

```
const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors');
const db = require('./db');

const app = express();
app.use(cors());
app.use(bodyParser.json());

const PORT = 3000;
app.listen(PORT, () => {
    console.log(`🚀 Serveur démarré sur http://localhost:${PORT}`);
});
```

4️⃣ Routes API RESTful

Ajoutons les routes pour **utilisateurs**, **articles** et **commentaires**.

📌 Gestion des utilisateurs

Dans `server.js`, ajouter :

```
// Récupérer tous les utilisateurs
app.get('/users', (req, res) => {
    db.query('SELECT * FROM users', (err, results) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json(results);
    });
});

// Récupérer un utilisateur par ID
app.get('/users/:id', (req, res) => {
    const { id } = req.params;
    db.query('SELECT * FROM users WHERE id = ?', [id], (err, results) => {
        if (err) return res.status(500).json({ error: err.message });
        if (results.length === 0) return res.status(404).json({ message:
"Utilisateur non trouvé" });
        res.json(results[0]);
    });
});

// Ajouter un utilisateur
app.post('/users', (req, res) => {
    const { name, email, password } = req.body;
    db.query('INSERT INTO users (name, email, password) VALUES (?, ?, ?)',
[name, email, password], (err, result) => {
        if (err) return res.status(500).json({ error: err.message });
        res.status(201).json({ id: result.insertId, name, email });
    });
});

// Mettre à jour un utilisateur
app.put('/users/:id', (req, res) => {
    const { id } = req.params;
    const { name, email } = req.body;
    db.query('UPDATE users SET name = ?, email = ? WHERE id = ?', [name,
email, id], (err) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json({ message: "Utilisateur mis à jour" });
    });
});

// Supprimer un utilisateur
app.delete('/users/:id', (req, res) => {
    const { id } = req.params;
    db.query('DELETE FROM users WHERE id = ?', [id], (err) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json({ message: "Utilisateur supprimé" });
    });
});
```

📌 Gestion des articles

Ajouter ces routes :

```
javascript
CopierModifier
// Récupérer tous les articles
app.get('/posts', (req, res) => {
    db.query('SELECT posts.*, users.name AS author FROM posts JOIN users ON
posts.user_id = users.id', (err, results) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json(results);
    });
});

// Récupérer un article par ID
app.get('/posts/:id', (req, res) => {
    const { id } = req.params;
    db.query('SELECT * FROM posts WHERE id = ?', [id], (err, results) => {
        if (err) return res.status(500).json({ error: err.message });
        if (results.length === 0) return res.status(404).json({ message:
"Article non trouvé" });
        res.json(results[0]);
    });
});

// Ajouter un article
app.post('/posts', (req, res) => {
    const { user_id, category_id, title, content } = req.body;
    db.query('INSERT INTO posts (user_id, category_id, title, content)
VALUES (?, ?, ?, ?)', [user_id, category_id, title, content], (err, result)
=> {
        if (err) return res.status(500).json({ error: err.message });
        res.status(201).json({ id: result.insertId, title, content });
    });
});

// Mettre à jour un article
app.put('/posts/:id', (req, res) => {
    const { id } = req.params;
    const { title, content } = req.body;
    db.query('UPDATE posts SET title = ?, content = ? WHERE id = ?',
[title, content, id], (err) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json({ message: "Article mis à jour" });
    });
});

// Supprimer un article
app.delete('/posts/:id', (req, res) => {
    const { id } = req.params;
    db.query('DELETE FROM posts WHERE id = ?', [id], (err) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json({ message: "Article supprimé" });
    });
});
```

📌 Gestion des commentaires

Ajouter ces routes :

```
// Récupérer les commentaires d’un article
app.get('/posts/:id/comments', (req, res) => {
    const { id } = req.params;
    db.query('SELECT * FROM comments WHERE post_id = ?', [id], (err,
results) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json(results);
    });
});

// Ajouter un commentaire
app.post('/posts/:id/comments', (req, res) => {
    const { id } = req.params;
    const { user_id, content } = req.body;
    db.query('INSERT INTO comments (post_id, user_id, content) VALUES (?,
?, ?)', [id, user_id, content], (err, result) => {
        if (err) return res.status(500).json({ error: err.message });
        res.status(201).json({ id: result.insertId, content });
    });
});
```

Tester une API RESTful avec Postman

1️⃣ Installer et Lancer Postman

- **Postman** est un outil qui permet d'envoyer des requêtes HTTP pour tester ton API.
- On commence par lancer notre serveur : node server.js
- On télécharge et installe **Postman** : https://www.postman.com/downloads/
- On ouvre Postman et crée un **nouvel onglet** pour envoyer des requêtes.

2️⃣ Tester les routes API

Notre API tourne sur **http://localhost:3000**.

On peut envoyer des **requêtes HTTP** à notre serveur.

A. Récupérer tous les utilisateurs

- **Méthode :** `GET`
- **URL :** `http://localhost:3000/users`
- **Action :** Récupère la liste des utilisateurs.
- **Résultat attendu :**

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

👉 **Cliquer sur "Send"** et vérifier la réponse JSON.

B. Ajouter un nouvel utilisateur

- **Méthode :** `POST`
- **URL :** `http://localhost:3000/users`
- **Onglet "Body" → Sélectionne "raw" → Choisis JSON**
- **Body :**

```
{
    "name": "Charlie",
```

```
    "email": "charlie@example.com",
    "password": "monmotdepasse"
}
```

- **Résultat attendu :**

```
{
    "id": 3,
    "name": "Charlie",
    "email": "charlie@example.com"
}
```

👉 **Cliquer sur "Send"** pour créer un utilisateur.

C. Modifier un utilisateur

- **Méthode :** `PUT`
- **URL :** `http://localhost:3000/users/3`
- **Body :**

```
{
    "name": "Charlie Dupont",
    "email": "charlie.dupont@example.com"
}
```

- **Résultat attendu :**

```
{
    "message": "Utilisateur mis à jour"
}
```

👉 **Envoie la requête pour modifier l'utilisateur 3.**

D. Supprimer un utilisateur

- **Méthode :** `DELETE`
- **URL :** `http://localhost:3000/users/3`
- **Résultat attendu :**

```
{
    "message": "Utilisateur supprimé"
}
```

👉 **Exécute la requête pour supprimer Charlie.**

3️⃣ Tester les articles

On peut tester de la même manière :

A. Récupérer tous les articles

- **Méthode :** `GET`
- **URL :** `http://localhost:3000/posts`

B. Créer un article

- **Méthode :** `POST`
- **URL :** `http://localhost:3000/posts`
- **Body :**

```
{
    "user_id": 1,
    "category_id": 1,
    "title": "Mon premier article",
    "content": "Ceci est le contenu de mon article"
}
```

4️⃣ Vérifier les erreurs

Si on envoie une requête incorrecte, on peut voir un **code d'erreur HTTP** dans Postman :

- **400 Bad Request** → Mauvais format de requête
- **404 Not Found** → L'élément demandé n'existe pas
- **500 Internal Server Error** → Problème dans le serveur

👉 Si une requête ne fonctionne pas, **vérifier le serveur Express.js** et les **logs de la console**.

🎯 Conclusion

✅ **Postman facilite les tests d'uneAPI** sans avoir besoin de frontend.
✅ **On utilise les méthodes HTTP (GET, POST, PUT, DELETE)** pour tester chaque route.
✅ **On regarde les logs dans la console Node.js** pour comprendre les erreurs éventuelles.