phpMyAdmin in Node.js: creare le API REST di base

phpMyAdmin ha reso popolare il concetto di un'applicazione web che permette di gestire un database MySQL fornendo un'interfaccia di amministrazione. Partendo da questo esempio, in questo articolo svilupperemo delle API con Node.js che ci consentiranno di svolgere delle operazioni essenziali su un'istanza di MySQL.

Requisiti

- Un'installazione di MySQL.
- Un'installazione di MongoDB.
- Node.js e NPM.

I moduli NPM richiesti

1. `express`
2. `body-parser`
3. `cors`
4. `dotenv`
5. `mongoose`
6. `mysql2`
7. `validator`
8. `jsonwebtoken`

Il file .env

Questo file di configurazione verrà usato dal modulo `dotenv` e servirà a fornire alla nostra applicazione alcuni parametri fondamentali per il suo funzionamento.

```
# .env

JWTSECRET=secret
JWTEXPIRESIN=604800
DBURI=mongodb://localhost:27017/mysql_client
HASHKEY=KvoF9FIYlg89DeiNbE4JOOlJl6GcQVTJ
HASHIV=8niXOZKtrJnSnXNC
```

1. `JWTSECRET`: La stringa privata usata per generare un JSON Web Token che consentirà l'accesso alle rotte protette.
2. `JWTEXPIRESIN`: Il numero di secondi che imposterà la scadenza del token di autenticazione.
3. `DBURI`: La stringa di connessione a MongoDB.

4. `HASHKEY`, `HASHIV`: Stringhe private per la cifratura e la decifratura delle password di connessione ai database MySQL.

Le collezioni di MongoDB

In MongoDB andremo a definire le due collezioni fondamentali per le nostre API.

1. `users`
2. `connections`

Con Mongoose definiremo il modello per la collezione `users` come segue:

```
// models/user.js

const { Schema, model } = require('mongoose');

const userSchema = new Schema({
    name: String,
    email: String,
    password: String,
    role: {
        type: String,
        default: 'user',
    },
    connections: [
        {
            type: Schema.Types.ObjectId,
            ref: 'Connection',
        },
    ]
});
module.exports = model('User', userSchema, 'users');
```

Il modello della collezione `connections` sarà invece il seguente:

```
// models/connection.js

const { Schema, model } = require('mongoose');

const connectionSchema = new Schema({
    host: String,
    port: String,
    username: String,
    password: String,
    database: String,
    user: {
        type: Schema.Types.ObjectId,
        ref: 'User',
    }
});

module.exports = model('Connection', connectionSchema, 'connections');
```

Un utente ha una relazione One-To-Many con le connessioni MySQL, rappresentate ciascuna da un ObjectID all'interno dell'array `connections` del modello `User`. Tramite il suo metodo `populate()`, Mongoose ci permetterà all'occorrenza di accedere ai vari documenti di tipo `Connection`.

Si tenga sempre presente che con Mongoose una stringa che rappresenta un ObjectID verrà sempre convertita e gestita nel formato richiesto da MongoDB.

Utility

Autenticazione

Per le funzioni di autenticazione abbiamo bisogno di una funzione middleware che convalidi il JSON Web Token passato con la richiesta e un'altra che lo generi con i dati dell'utente che ha effettuato il login.

```
// utils/auth.js

const jwt = require('jsonwebtoken');

const authMiddleware = (req, res, next) => {
    const header = req.headers.authorization;
    const token = header && header.split(' ')[1];
    if (!token) {
        return res.status(401).json({ message: 'Token is required' });
    }
    jwt.verify(token, process.env.JWTSECRET, (err, decoded) => {
        if (err) {
            return res.status(401).json({ message: 'Invalid token' });
        }
        req.user = decoded;
        next();
    });
};

const generateToken = (data) => {
    const expires = parseInt(process.env.JWTEXPIRESIN, 10);
    const plainData = { name: data.name, email: data.email };
    try {
        return jwt.sign(plainData, process.env.JWTSECRET, { expiresIn: expires });
    } catch (error) {
        return '';
    }
};

module.exports = { authMiddleware, generateToken };
```

Per generare un token con successo, è di fondamentale importanza che l'oggetto JavaScript contenga solo le proprietà essenziali, ossia sia di tipo **plain**. In questo caso l'oggetto `plainData` contiene solo il nome e l'email dell'utente.

Cifratura e hashing

Per cifrare la password della connessione ad un database MySQL e quindi decifrarla all'occorrenza, possiamo usare il modulo core `crypto` e l'algoritmo `aes-256-cbc`. L'hashing della password di un utente sfrutterà invece l'algoritmo SHA-256.

```
// utils/hash.js

const crypto = require('crypto');
const algorithm = 'aes-256-cbc';
const key = Buffer.from(process.env.HASHKEY);
const iv = Buffer.from(process.env.HASHIV);

const encrypt = (text) => {
    const cipher = crypto.createCipheriv(algorithm, key, iv);
    let encrypted = cipher.update(text, 'utf8', 'hex');
    encrypted += cipher.final('hex');
    return encrypted;
};

const decrypt = (encryptedText) => {
    const decipher = crypto.createDecipheriv(algorithm, key, iv);
    let decrypted = decipher.update(encryptedText, 'hex', 'utf8');
    decrypted += decipher.final('utf8');
    return decrypted;
};

const hash = (password) => {
    return crypto.createHash('sha256').update(password).digest('hex');
};

module.exports = { encrypt, decrypt, hash };
```

MySQL

Il modulo `mysql2` permette di gestire pool di connessioni. Questa caratteristica è indicata nei casi in cui abbiamo bisogno di connetterci efficientemente a più di un database, come nel nostro caso in cui un utente può salvare più connessioni.

```
// utils/mysql.js

const mysql = require('mysql2/promise');
const User = require('../models/user');
const { decrypt } = require('./hash');

const getUserConnectionDetails = async (userId, databaseName) => {
    try {
        const user = await User.findById(userId).populate('connections').exec();
        const connections = user.connections.filter((connection) =>
connection.database === databaseName);
        return connections.length > 0 ? connections[0] : null;
    } catch (error) {
        return null;
    }
};
```

```
const createMySQLConnectionPool = async (userId, databaseName) => {
    const connectionDetails = await getUserConnectionDetails(userId, databaseName);
    if (!connectionDetails) {
        return null;
    }

    const { host, port, username, password, database } = connectionDetails;
    const decryptedPassword = decrypt(password);
    return mysql.createPool({
        host,
        port,
        user: username,
        password: decryptedPassword,
        database,
    });
};

const executeQueryForUser = async (userId, databaseName, query) => {
    const pool = await createMySQLConnectionPool(userId, databaseName);
    if (!pool) {
        return null;
    }

    const [rows] = await pool.query(query);
    pool.end();
    return rows;
};

module.exports = { executeQueryForUser };
```

Come si può notare, dobbiamo innanzitutto recuperare la connessione che corrisponde al database scelto dall'utente. Il pool della connessione viene creato dopo aver effettuato la decifratura della password. La funzione esportata, `executeQueryForUser(userId, databaseName, query)` esegue una query sul database selezionato e restituisce le righe trovate come risultato.

Le API

```
// routes/index.js

const express = require('express');
const router = express.Router();
const userController = require('../controllers/users');
const connectionController = require('../controllers/connections');
const auth = require('../utils/auth');

router.post('/users/create', userController.createUser);
router.post('/users/login', userController.loginUser);
router.put('/users/update/:id', auth.authMiddleware, userController.updateUser);
router.delete('/users/delete/:id', auth.authMiddleware, userController.deleteUser);

router.get('/connections/:id/tables', auth.authMiddleware,
connectionController.listTablesInUserDatabase);
```

```
router.get('/connections/:id/data', auth.authMiddleware,
connectionController.listColumnsAndDataInTable);
router.post('/connections/create', auth.authMiddleware,
connectionController.createConnection);

module.exports = router;
```

CRUD User

Le operazioni CRUD (Create,Read,Update,Delete) non presentano particolare interesse e non ci soffermeremo su di esse in questo articolo, limitandoci a riportarle per completezza di informazioni.

```
// controllers/users.js

const { hash } = require('../utils/hash');
const auth = require('../utils/auth');
const User = require('../models/user');

const createUser = async (req, res, next) => {
    try {
        const { name, email, password } = req.body;
        const hashedPassword = hash(password);
        const user = new User({
            name,
            email,
            password: hashedPassword
        });
        await user.save();
        return res.status(201).json(user);
    } catch (error) {
        return res.status(500).json({ error });
    }
};

const loginUser = async (req, res, next) => {
    try {
        const { email, password } = req.body;
        const hashedPassword = hash(password);
        const user = await User.findOne({ email, password: hashedPassword });
        if (!user) {
            return res.status(200).json({ error: 'Invalid credentials' });
        }
        const token = auth.generateToken(user);
        if (!token) {
            return res.status(200).json({ error: 'Token generation failed' });
        }
        return res.status(200).json({ token, email: user.email, name: user.name });
    } catch (error) {
        return res.status(500).json({ error });
    }
};

const updateUser = async (req, res, next) => {
    try {
```

```
        const id = req.params.id;
        const body = req.body;
        let update = {};
        if (body.name) {
            update.name = body.name;
        }
        if (body.email) {
            update.email = body.email;
        }
        if (body.password) {
            update.password = hash(body.password);
        }
        const user = await User.findByIdAndUpdate(id, update, { new: true });
        if (!user) {
            return res.status(200).json({ error: 'User not found' });
        }
        return res.status(200).json(user);
    } catch (error) {
        return res.status(500).json({ error });
    }
};

const deleteUser = async (req, res, next) => {
    try {
        const id = req.params.id;
        const user = await User.findByIdAndDelete(id);
        if (!user) {
            return res.status(200).json({ error: 'User not found' });
        }
        return res.status(200).json(user);
    } catch (error) {
        return res.status(500).json({ error });
    }
};


module.exports = {
    createUser,
    loginUser,
    updateUser,
    deleteUser
};
```

Operazioni su MySQL

Il primo passo per operare su MySQL è quello di creare una nuova connessione e associarla all'utente che ne ha inserito i parametri.

```
// controllers/connections.js

const Connection = require('../models/connection');
const validator = require('validator');
const { encrypt } = require('../utils/hash');
const { executeQueryForUser } = require('../utils/mysql');
```

```
const User = require('../models/user');

const createConnection = async (req, res, next) => {
    const { host, port, username, password, database } = req.body;
    if (!host || !port || !username || !password || !database) {
        return res.status(200).json({ error: 'All fields are required' });
    }
    if(host !== 'localhost') {
        if(!validator.isIP(host)) {
            return res.status(200).json({ error: 'Invalid host' });
        }
    }
    if(!validator.isPort(port)) {
        return res.status(200).json({ error: 'Invalid port' });
    }
    const encryptedPassword = encrypt(password);
    const connection = new Connection({
        host,
        port,
        username,
        password: encryptedPassword,
        database,
        user: req.body.id
    });
    try {
        const conn = await connection.save();
        const user = await User.findById(req.body.id);
        user.connections.push(conn._id);
        await user.save();
        return res.status(201).json({ message: 'Connection created' });
    } catch (err) {
        return res.status(500).json({ error: err });
    }
};
```

I dati in ingresso vengono validati dal modulo `validator`. Se il parametro `host` è diverso da `localhost`, viene qui richiesto un indirizzo IP valido. Allo stesso modo viene richiesto un numero valido per il parametro `port`, dove per valido si intende un valore numerico compreso nel range di porte standard.

Qui notiamo una limitazione nel nostro codice: non sono di fatto ammessi nomi di host diversi da `localhost` e questo potrebbe essere un problema se volessimo connetterci ad un server remoto che, per via della sua configurazione, non accetta un indirizzo IP come parametro.

Soffermiamoci un istante sulla creazione della connessione e sul collegamento alla collection `users`:

```
const conn = await connection.save();
const user = await User.findById(req.body.id);
user.connections.push(conn._id);
await user.save();
```

Mongoose enfatizza nella sua documentazione l'uso del metodo `save()` sia in fase di creazione che di aggiornamento di un documento. Qui `conn` ci permette di accedere all'ObjectID del documento

appena creato. Allo stesso modo, invece di usare l'approccio con `$push` per inserire l'ObjectID nell'array `connections`, possiamo modificare l'istanza `user` usando un semplice approccio JavaScript e quindi invocare `save()` per salvare la modifica.

Definiamo ora un metodo per elencare le tabelle presenti nel database scelto dall'utente.

```
// controllers/connections.js

const listTablesInUserDatabase = async (req, res, next) => {
    const { database } = req.query;
    if (!database) {
        return res.status(200).json({ error: 'Database name is required' });
    }
    const query = 'SHOW TABLES';
    const tables = await executeQueryForUser(req.params.id, database, query);
    const tableNames = tables.map((table) => table[`Tables_in_${database}`]);
    if (!tables) {
        return res.status(500).json({ error: 'Failed to fetch tables' });
    }
    return res.status(200).json({ tables: tableNames });
};
```

La query MySQL che stiamo eseguendo è `SHOW TABLES` che nell'implementazione di `mysql2` restituisce un array di oggetti contenenti una sola proprietà identica per tutti (`Tables_in_${database}`) e su cui dobbiamo applicare `map()` per trasformarlo in un array lineare di nomi di tabelle.

Come ultimo passaggio, possiamo elencare le colonne ed i dati presenti in una tabella scelta dall'utente.

```
// controllers/connections.js

const listColumnsAndDataInTable = async (req, res, next) => {
    const { database, table } = req.query;
    if (!database || !table) {
        return res.status(200).json({ error: 'Database and table names are
required' });
    }
    const page = req.query.page ? parseInt(req.query.page, 10) : 1;
    const perPage = 10;
    const offset = (page - 1) * perPage
    const queryColumns = `SHOW COLUMNS FROM ${table}`;
    const columns = await executeQueryForUser(req.params.id, database,
queryColumns);
    const query = `SELECT * FROM ${table} LIMIT ${offset}, ${perPage}`;
    const data = await executeQueryForUser(req.params.id, database, query);
    if (!data) {
        return res.status(500).json({ error: 'Failed to fetch data' });
    }
    return res.status(200).json({ columns, data });
};
```

```
module.exports = { createConnection, listTablesInUserDatabase,
listColumnsAndDataInTable };
```

La query `SHOW COLUMNS FROM ${table}` mostra metadati importanti che possiamo mostrare all'utente. Ad esempio:

```
const columns = [
 {
  "Field": "id",
  "Type": "int",
  "Null": "NO",
  "Key": "PRI",
  "Default": null,
  "Extra": "auto_increment"
 },
 {
  "Field": "title",
  "Type": "varchar(255)",
  "Null": "NO",
  "Key": "",
  "Default": null,
  "Extra": ""
 },
 //...
];
```

Come si può notare, abbiamo tutti i dati necessari per realizzare nel frontend una view che mostri la tabella selezionata.

L'applicazione

```
// app.js

require('dotenv').config();

const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors');
const mongoose = require('mongoose');
const PORT = process.env.PORT || 3000;
const routes = require('./routes');

mongoose.connect(process.env.DBURI);

const app = express();

app.use(cors());
app.use(bodyParser.json());

app.use('/api', routes);

app.listen(PORT);
```

È possibile passare all'applicazione il numero di porta su cui creare l'istanza semplicemente aggiungendo la variabile `PORT` al file `.env`.

Conclusione

Abbiamo implementato i fondamenti da cui possiamo estendere la nostra applicazione andando in futuro ad aggiungere altri endpoint con cui gestire i nostri database MySQL. Sicuramente è un'ottimo modo per studiare gli aspetti core di MySQL a cui di solito siamo abituati a pensare come a qualcosa di disponibile solo dalla console.

Applicazioni Correlate

- 

  ### Node.js Placeholder Image

  Applicazione per la generazione con Node.js di immagini segnaposto.
  DockerDocker ComposeNode.jsJavaScriptExpressJS

- 

Node.js URL Shortener

Implementazione in Node.js di un sistema per l'abbreviazione degli URL.
DockerDocker ComposeNode.jsJavaScriptExpressJSMongoDB

- ### JavaScript App Hash Change

  Applicazione che sfrutta gli hash degli URL per gestire contenuto dinamico in JavaScript. DockerDocker ComposeNode.jsJavaScriptExpressJSMySQL