TP : Création d'une api RESTful avec Node.js et PostgreSQL

En tant que développeur Web moderne, il est extrêmement important de savoir utiliser des API pour faciliter la communication entre différents systèmes logiciels.

Dans ce tutorial, vous apprendrez à créer votre propre API RESTful dans un environnement Node.js exécuté sur un serveur Express et utilisant une base de données PostgreSQL.

Prérequis

Pour tirer le meilleur parti de ce tutorial, vous devez:

- Vous devez être familiarisé avec la syntaxe et les principes fondamentaux de JavaScript.
- Vous devez avoir des connaissances de base pour travailler en lignes de commandes
- Vous devriez avoir Node.js et npm installés

Objectifs

A la fin de ce tp, vous allez disposer d'un serveur d'API RestFul entièrement fonctionnel qui s'exécute dans Node.js et utilisant le framwork Express. L'API doit être capable de gérer les méthodes de requête HTTP correspondant à la base de données PostgreSQL à partir de laquelle l'API récupère ses données. Vous allez apprendre à l'utiliser Postgresql avec l'interface de ligne de commande.

Qu'est-ce qu'une API RESTful?

REST signifie (**Re**presentational **S**tate **T**ransfer) et définit un ensemble de normes pour les services Web. Une API est une interface utilisée par différents logiciels pour communiquer entre elles. Par conséquent, une API RESTful est une API conforme à l'architecture REST et ses contraintes. Les systèmes REST sont sans état, évolutifs, peuvent être mis en cache et possèdent une interface uniforme.

Les API RESTful utilisent le plus souvent des requêtes HTTP. Les quatre méthodes HTTP les plus courantes sont `GET` , `POST` , `PUT` , et `DELETE` , ses méthodes permettant au développeur de créer un système CRUD: Create, Read, Update, Delete.

Base de données PostgreSQL

PostgreSQL, souvent appelé Postgres, est un système de gestion de base de données relationnelle gratuit et Open source. Vous connaissez peut-être quelques autres systèmes de base de données similaires, tels que MySQL, Microsoft SQL Server ou MariaDB, qui font concurrence à PostgreSQL.

PostgreSQL est une base de données relationnelle robuste mais stable qui existe depuis 1997 et qui est disponible sur tous les principaux systèmes d'exploitation - Linux, Windows et macOS. Étant donné que PostgreSQL est réputé pour sa stabilité, son extensibilité et sa conformité aux normes, il s'agit d'un choix très prisé des développeurs et des entreprises pour répondre à leurs besoins en bases de données.

Nous allons commencer ce tutoriel par la création d'un nouvel utilisateur, la création d'une base de données, la création d'une nouvelle table et l'initialisation de certaines données.

Invite de commande PostgreSQL

`psql` est un client PostgreSQL en ligne de commande. L'exécution de la commande `psql` vous connectera au serveur PostgreSQL. Exécuter `psql --help` vous donnera plus d'informations sur les options disponibles pour vous connecter à `psql` .

- `-h` — `--host=HOSTNAME` | adresse du serveur base de données
- `-p` — `--port=PORT` | le port du serveur de la base de données)
- `-U` — `--username=USERNAME` | nom d'utilisateur de la base de données
- `-w` — `--no-password` | ne jamais demander le mot de passe
- `-W` — `--password` | forcer la demande du mot de passe avant de se connecter.

Nous nous contenterons à la base de données par défaut `postgres` avec les informations de connexion par défaut - sans option.

```
1   psql postgres
```

INSY2S

Vous verrez que nous avons établi une nouvelle connexion. Nous sommes maintenant connecter à la base de données `postgres` via le client `psql` L'invite se termine par un `#` pour indiquer que nous sommes connectés en tant que superutilisateur, ou root.

```
postgres=#
```

INSY2S

Les commandes de base de `psql` commencent par un backslash (`\`). Pour tester notre première commande, nous pouvons vérifier à quelle base de données, utilisateur et port nous nous sommes connectés en utilisant la commande `\conninfo` .

```
postgres=# \conninfo
You are connected to database "postgres" as user "your_username" via socket in "/tmp"
```

INSY2S

Voici quelques commandes courantes que nous utiliserons dans ce tutoriel.

- `\q` | Quitter la connexion `psql`
- `\c` | Se connecter à une nouvelle base de données
- `\dt` | Lister toutes les tables
- `\du` | Lister tous les rôles
- `\list` | Liste des bases de données

Créons une nouvelle base de données et un nouvel utilisateur pour ne pas utiliser les comptes par défaut, dotés des privilèges du superutilisateur.

Créer un utilisateur

Premièrement, nous allons créer un rôle appelé `me` et lui données le mot de passe `password` . Un rôle peut fonctionner en tant qu'utilisateur ou groupe. Dans ce cas, nous l'utilisons en tant qu'utilisateur.

```
postgres=# CREATE ROLE me WITH LOGIN PASSWORD 'password';
```

INSY2S

Nous voulons que `me` puisse créer une base de données.

```
postgres=# ALTER ROLE me CREATEDB;
```

INSY2S

Vous pouvez exécuter `\du` pour lister tous les rôles/utilisateurs.

```
me          | Create DB                          | {}
postgres    | Superuser, Create role, Create DB  | {}
```

INSY2S

Maintenant, nous voulons créer une base de données à partir de l'utilisateur `me` "ME". Quittez la session par défaut avec `\q`

```
postgres=# \q
```

INSY2S

Nous sommes retournés au terminal par défaut de notre ordinateur. Nous allons maintenant se connecter à `postgres` avec l'utilisateur `me` .

```
psql -d postgres -U me
```

INSY2S

Au lieu de `postgres=#` , notre invite affiche `postgres=>` , ce qui signifie que nous ne sommes plus connectés en tant que superutilisateur.

Créer une base de données

Nous pouvons créer une base de données avec la commande SQL.

```
postgres=> CREATE DATABASE api;
```

INSY2S

Utilisez la commande `\list` pour voir les bases de données disponibles.

```
Name    |    Owner    | Encoding |   Collate   |    Ctype    |
api     | me          | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
```

INSY2S

Nous allons nous connecter à la nouvelle base de données `api` avec l'utilisateur `me` à l'aide de la commande `\c` (connect).

```
postgres=> \c api
You are now connected to database "api" as user "me".
api=>
```

INSY2S

Notre invite de commandes indique maintenant que nous sommes connectés à la base de données. `api` .

Créer une table

La dernière chose que nous ferons dans l'invite de commande `psql` la création d'une table appelée `users` avec trois champs - deux de type `VARCHAR` et un identifiant `PRIMARY KEY` auto-incrémenté.

```
api=>
CREATE TABLE users (
  ID SERIAL PRIMARY KEY,
  name VARCHAR(30),
  email VARCHAR(30)
);
```

INSY2S

Nous ajouterons deux utilisateurs dans la tables users `users` pour que nous puissions utiliser certaines données.

```
INSERT INTO users (name, email)
  VALUES ('Jerry', 'jerry@example.com'), ('George', 'george@example.com');
```

INSY2S

Nous allons vérifer que cela a été ajouté correctement en obtenant toutes les lignes de la table `users` .

```
api=> SELECT * FROM users;
id |  name  |       email
----+--------+--------------------
  1 | Jerry  | jerry@example.com
  2 | George | george@example.com
```

INSY2S

Nous avons maintenant un utilisateur, une base de données, une table et des données. Nous pouvons commencer à construire notre API RESTful avec Node.js pour exposer les données stockées dans notre base de données PostgreSQL.

Configuration d'un serveur "Express"

À ce stade, nous avons terminé toutes nos tâches PostgreSQL et nous pouvons commencer à configurer notre application Node.js et notre serveur Express.

Créez un répertoire pour notre projet.

```
mkdir node-api-postgres
cd node-api-postgres
```

INSY2S

Vous pouvez exécuter la commande `npm init -y` pour créer le fichier de configuration `package.json` automatiquement, ou bien copier le code ci-dessous dans un fichier `package.json` à la racine de votre projet.

```
{
  "name": "node-api-postgres",
  "version": "1.0.0",
  "description": "RESTful API with Node.js, Express, and PostgreSQL",
  "main": "index.js",
  "license": "MIT"
}
```

INSY2S

Nous allons installer Express pour créer le serveur et node-postgres (pg) pour pouvoir se connecter à PostgreSQL.

```
npm i express pg
```

INSY2S

Nous avons maintenant nos dépendances chargées dans le répertoire `node_modules` et le fichier de configuration `package.json` .

Créez un fichier `index.js` , que nous utiliserons comme point d'entrée pour notre serveur. En haut du fichier, nous aurons besoin du module `express` module, déclarer le `bodyParser` et configurer nos variables `app` et `port`

```
const express = require('express')
const bodyParser = require('body-parser')
const app = express()
const port = 3000

app.use(bodyParser.json())
app.use(
  bodyParser.urlencoded({
    extended: true,
  })
)
```

INSY2S

Nous allons mapper une requête . `GET` sur l'URL racine (`/`) pour renvoyer un flux JSON

```
app.get('/', (request, response) => {
  response.json({ info: 'Node.js, Express, and Postgres API' })
})
```

INSY2S

Maintenant, configurez l'application pour qu'elle écoute sur le port que vous avez défini.

```
app.listen(port, () => {
  console.log(`App running on port ${port}.`)
})
```

INSY2S

À partir de la ligne de commande `node` , nous pouvons démarrer le serveur en précisant le fichier `index.js` .

```
node index.js
App running on port 3000.
```

INSY2S

Accédez à `http://localhost:3000` dans la barre d'URL de votre navigateur et vous verrez le JSON que nous avons défini précédemment.

```
{
  info: "Node.js, Express, and Postgres API"
}
```

INSY2S

Le serveur Express est démarré, mais il n'envoie que certaines données JSON statiques que nous avons créées. L'étape suivante consiste à vous connecter à PostgreSQL à partir de Node.js pour pouvoir effectuer des requêtes dynamiques.

Connexion à la base de données à partir de Node.js

Nous allons utiliser le module node-postgres pour créer un pool de connexions. De cette façon, nous n'avons pas besoin d'ouvrir un client et de le fermer à chaque fois que nous faisons une requête.

Créez un fichier appelé `queries.js` et y ajouter la configuration de votre connexion PostgreSQL.

```
const Pool = require('pg').Pool
const pool = new Pool({
  user: 'me',
  host: 'localhost',
  database: 'api',
  password: 'password',
  port: 5432,
})
```

INSY2S

Dans un environnement de production, vous devriez placer votre configuration dans un fichier séparé avec des autorisations restrictives qui ne sont pas accessibles à partir de votre logiciel de gestion de version, mais pour la simplicité de ce Tutorial, nous le conservons dans le même fichier que les requêtes.

L'objectif de ce Tutorial est d'autoriser les opérations `GET` , `POST` , `PUT` , et `DELETE` sur l'API qui exécutera les commandes de base de données correspondantes. Pour ce faire, nous allons configurer une route pour chaque noeud final et une fonction correspondant à chaque requête.

Créer des routes (urls)

Nous allons créer six fonctions pour six routes, comme indiqué ci-dessous. Tout d'abord, nous allons passer en revue et créer toutes les fonctions pour chaque route, puis nous allons exporter les fonctions afin qu'elles soient accessibles:

- `GET` — `/` | `displayHome()`
- `GET` — `/users` | `getUsers()`
- `GET` — `/users/:id` | `getUserById()`
- `POST` — `users` | `createUser()`
- `PUT` — `/users/:id` | `updateUser()`
- `DELETE` — `/users/:id` | `deleteUser()`

Dans le fichier `index.js` , nous avons créé un service avec l'instruction `app.get()` qui lie l'url racine (`/`) à une fonction définie dedans. Maintenant, nous allons créer des points les service qui afficheront tous les utilisateurs, afficheront un seul utilisateur par son id , créerons un nouvel utilisateur, mettront à jour un utilisateur existant et supprimeront un utilisateur. `queries.js` pour le point de terminaison racine contenant une fonction. Maintenant, dans

Obtenir tous les utilisateurs

Notre premier service sera en méthode `GET` . Dans la fonction `pool.query()` nous pouvons mettre le SQL brut qui va être executer dans la base de données `api` . Nous allons sélectionner (`SELECT`) tous les utilisateurs et les trier par identifiant.

```
const getUsers = (request, response) => {
  pool.query('SELECT * FROM users ORDER BY id ASC', (error, results) => {
    if (error) {
      throw error
    }
    response.status(200).json(results.rows)
  })
}
```

INSY2S

Obtenir un seul utilisateur par id

Pour notre requête `/users/:id` , nous allons obtenir le paramètre `id` personnalisé de l'URL et l'utiliser dans une clause `WHERE` pour afficher le résultat.

Dans la requête SQL, nous recherchons les lignes qui vérifient la condition `id=$1` . Dans cet exemple `$1` est le premier paramétre d'entré.

```
const getUserById = (request, response) => {
  const id = parseInt(request.params.id)

  pool.query('SELECT * FROM users WHERE id = $1', [id], (error, results) => {
    if (error) {
      throw error
    }
    response.status(200).json(results.rows)
  })
}
```

INSY2S

Ajouter un nouvel utilisateur

L'API gérera deux services `GET` et `POST` en utilisant le même url `/users` . En `POST` nous allons ajouter un nouvel utilisateur. Dans cette fonction, nous extrayons les propriétés `name` et `email` du corps (bodu) http de la requêtes et insérons (`INSERT`) les valeurs en base de données en sql.

```
const createUser = (request, response) => {
  const { name, email } = request.body

  pool.query('INSERT INTO users (name, email) VALUES ($1, $2)', [name, email], (error, results) => {
    if (error) {
      throw error
    }
    response.status(201).send(`User added with ID: ${result.insertId}`)
  })
}
```

INSY2S

Mettre à jour un utilisateur existant

L'url `/users/:id` prendra également deux requêtes HTTP : Le `GET` que nous avons créé pour `getUserById` , et un `PUT` , pour modifier un utilisateur existant. Pour cette requête, nous allons combiner ce que nous avons appris dans `GET` et `POST` pour utiliser la clause `UPDATE`

Il est à noter que `PUT` est **idempotent**, ce qui signifie que le même appel peut être répété encore et encore et produira le même résultat. En `POST` , le même appel répété créera continuellement de nouveaux utilisateurs avec les mêmes données.

```
const updateUser = (request, response) => {
  const id = parseInt(request.params.id)
  const { name, email } = request.body

  pool.query(
    'UPDATE users SET name = $1, email = $2 WHERE id = $3',
    [name, email, id],
    (error, results) => {
      if (error) {
        throw error
      }
      response.status(200).send(`User modified with ID: ${id}`)
    }
  )
}
```

INSY2S

Supprimer un utilisateur

Enfin, nous allons utiliser la clause `DELETE` pour l'url `/users/:id` afin de supprimer un utilisateur spécifique par son identifiant. Cet appel est très similaire à notre fonction `getUserById()`

```
1  const deleteUser = (request, response) => {
2    const id = parseInt(request.params.id)
3
4    pool.query('DELETE FROM users WHERE id = $1', [id], (error, results) => {
5      if (error) {
6        throw error
7      }
8      response.status(200).send(`User deleted with ID: ${id}`)
9    })
10 }
```

INSY2S

Exportation

Pour accéder à ces fonctions à partir de `index.js` en utilisant `module.exports` , en créant un objet à partir d'une fonctions. Comme nous utilisons la syntaxe ES6, nous pouvons écrire `getUsers` au lieu de `getUsers:getUsers` , etc.

```
1  module.exports = {
2    getUsers,
3    getUserById,
4    createUser,
5    updateUser,
6    deleteUser,
7  }
```

INSY2S

Voici notre fichier `queries.js` complet.

```
const Pool = require('pg').Pool
const pool = new Pool({
  user: 'me',
  host: 'localhost',
  database: 'api',
  password: 'password',
  port: 5432,
})
const getUsers = (request, response) => {
  pool.query('SELECT * FROM users ORDER BY id ASC', (error, results) => {
    if (error) {
      throw error
    }
    response.status(200).json(results.rows)
  })
}

const getUserById = (request, response) => {
  const id = parseInt(request.params.id)

  pool.query('SELECT * FROM users WHERE id = $1', [id], (error, results) => {
    if (error) {
      throw error
    }
    response.status(200).json(results.rows)
  })
}

const createUser = (request, response) => {
  const { name, email } = request.body

  pool.query('INSERT INTO users (name, email) VALUES ($1, $2)', [name, email], (
error, results) => {
    if (error) {
      throw error
    }
    response.status(201).send(`User added with ID: ${result.insertId}`)
  })
}

const updateUser = (request, response) => {
  const id = parseInt(request.params.id)
  const { name, email } = request.body

  pool.query(
    'UPDATE users SET name = $1, email = $2 WHERE id = $3',
    [name, email, id],
    (error, results) => {
```

```
    (error, results) => {
      if (error) {

      }
      response.status(200).send(`User modified with ID: ${id}`)
    }
  )
}

const deleteUser = (request, response) => {
  const id = parseInt(request.params.id)

  pool.query('DELETE FROM users WHERE id = $1', [id], (error, results) => {
    if (error) {
      throw error
    }
    response.status(200).send(`User deleted with ID: ${id}`)
  })
}

module.exports = {
  getUsers,
  getUserById,
  createUser,
  updateUser,
  deleteUser,
}
```

INSY2S

```
throw error
```

Completing the setup

Maintenant que nous avons toutes nos requêtes, la dernière chose à faire est de les importer dans le fichier `index.js` et ajouter les urls routes pour fonctions de requêtes que nous avons créé.

Pour obtenir toutes les fonctions exportées à partir de `queries.js` , nous avons besoin exporter le fichier en l'affectant à une variable.

```
1  const db = require('./queries')
```

INSY2S

Désormais, pour chaque service, nous allons définir la méthode HTTP, le chemin d'URL et la fonction correspondante.

```
1  app.get('/users', db.getUsers)
2  app.get('/users/:id', db.getUserById)
3  app.post('/users', db.createUser)
4  app.put('/users/:id', db.updateUser)
5  app.delete('/users/:id', db.deleteUser)
```

INSY2S

Voici notre fichier `index.js` complet, le point d'entrée du serveur API.

```
const express = require('express')
const bodyParser = require('body-parser')
const app = express()
const db = require('./queries')
const port = 3000

app.use(bodyParser.json())
app.use(
  bodyParser.urlencoded({
    extended: true,
  })
)

app.get('/', (request, response) => {
  response.json({ info: 'Node.js, Express, and Postgres API' })
})

app.get('/users', db.getUsers)
app.get('/users/:id', db.getUserById)
app.post('/users', db.createUser)
app.put('/users/:id', db.updateUser)
app.delete('/users/:id', db.deleteUser)

app.listen(port, () => {
  console.log(`App running on port ${port}.`)
})
```

INSY2S

Maintenant, avec seulement ces deux fichiers, nous avons un serveur, une base de données et une API tous configurés. Vous pouvez démarrer le serveur en untilisant à nouveau la commande `node` et en précisant le fichier. `index.js` .

```
node index.js
App running on port 3000.
```

style="visibility: hidden;"

Maintenant, si vous entrer l'url `http://localhost:3000/users` ou l'url `http://localhost:3000/users/1` sur un navigateur, vous verrez la réponse JSON des deux requêtes `GET` . Mais comment pouvons-nous tester nos demandes `POST` , `PUT` , et `DELETE` ?

Cela peut être fait avec curl, un outil de ligne de commande déjà disponible sur votre terminal ou via postman. Vous trouverez ci-dessous des exemples que vous pouvez exécuter sur la ligne de commande pour tester tous les protocoles avec curl.

> *Assurez-vous que le serveur est lancé depuis le terminal pendant que vous exécutez ces commandes dans une fenêtre séparé*

POST

Ajouter un nouvel utilisateur avec le `name` Elaine et `l'email` elaine@example.com.

```
curl --data "name=Elaine&email=elaine@example.com"
http://localhost:3000/users
```

INSY2S

PUT

Mettre à jour l'utilisateur dont l'id est `1` pour avoir le `name` Kramer et `l'email` kramer@example.com.

```
curl -X PUT -d "name=Kramer" -d "email=kramer@example.com"
http://localhost:3000/users/1
```

INSY2S

DELETE

Supprimer l'utilisateur dont l'id est `1` .

```
curl -X "DELETE" http://localhost:3000/users/1
```

INSY2S

Conclusion

Félicitations, vous devriez maintenant avoir un serveur d'API fonctionnel fonctionnant sur Node.js et connecté à une base de données PostgreSQL active. Dans ce Tutorial, nous avons appris à installer et à configurer PostgreSQL en lignes de commandes, à créer des utilisateurs, des bases de données et des tables, ainsi qu'à exécuter des commandes SQL. Nous avons également appris à créer un serveur Express pouvant gérer plusieurs méthodes HTTP et à utiliser le module `pg` pour se connecter à PostgreSQL depuis Node.js

Avec ces connaissances, vous devriez pouvoir utiliser cette API et l'utiliser pour vos projets de développement personnel ou professionnel.