TP - Système de Gestion Utilisateurs avec PostgreSQL et Node.js

Rôles, Permissions et Authentification

IUT Nord Franche-Comté
Objectif: Créer un système complet de gestion utilisateurs avec authentification, rôles et permissions

---

📋 Contexte

Vous êtes développeur dans une ESN et devez créer le backend d'un système de gestion d'utilisateurs pour une application web. Le système doit gérer l'authentification, les rôles, les permissions et garder un historique des connexions.

🎯 Objectifs d'Apprentissage

À la fin de ce TP, vous saurez:

- ✅ Modéliser une base de données avec relations Many-to-Many
- ✅ Créer des tables avec contraintes et relations
- ✅ Implémenter des transactions SQL
- ✅ Utiliser la librairie `pg` avec Node.js
- ✅ Créer une API REST avec Express
- ✅ Gérer l'authentification avec tokens
- ✅ Implémenter un système de permissions

Partie 1 : Mise en Place de la Base de Données

Exercice 1.1 : Créer la Base de Données

```
# Se connecter à PostgreSQL
psql -U postgres
```

```
# Créer la base de données
CREATE DATABASE gestion_utilisateurs;

# Se connecter à la base
\c gestion_utilisateurs
```

Exercice 1.2 : Créer les Tables Principales

📝 **Task 1:** Créez la table `utilisateurs`

```
CREATE TABLE utilisateurs (
    -- À COMPLÉTER
    -- Colonnes nécessaires:
    -- - id (auto-incrémenté, clé primaire)
    -- - email (unique, obligatoire, format email)
    -- - password_hash (obligatoire)
    -- - nom (optionnel)
    -- - prenom (optionnel)
    -- - actif (booléen, défaut: true)
    -- - date_creation (timestamp, défaut: maintenant)
    -- - date_modification (timestamp, défaut: maintenant)
);

-- Index pour recherche rapide
CREATE INDEX idx_utilisateurs_email ON utilisateurs(email);
CREATE INDEX idx_utilisateurs_actif ON utilisateurs(actif);
```

💡 **Indice:** Utilisez `SERIAL` pour l'auto-incrémentation, `UNIQUE` pour éviter les doublons d'email, et `CHECK` pour valider le format email.

📝 **Task 2:** Créez les tables `roles` et `permissions`

```
CREATE TABLE roles (
    -- À COMPLÉTER
    -- id, nom (unique), description, date_creation
);

CREATE TABLE permissions (
    -- À COMPLÉTER
    -- id, nom (unique), ressource, action, description
```

```
    -- BONUS: Ajoutez une contrainte UNIQUE sur (ressource, action)
);
```

Exercice 1.3 : Tables d'Association (Many-to-Many)

📝 **Task 3:** Créez les tables d'association pour gérer les relations N-N

```
-- Table association utilisateur_roles
CREATE TABLE utilisateur_roles (
    -- À COMPLÉTER
    -- utilisateur_id (FK vers utilisateurs)
    -- role_id (FK vers roles)
    -- date_assignation
    -- Clé primaire composite
    -- ON DELETE CASCADE
);

-- Table association role_permissions
CREATE TABLE role_permissions (
    -- À COMPLÉTER
    -- role_id (FK vers roles)
    -- permission_id (FK vers permissions)
    -- Clé primaire composite
    -- ON DELETE CASCADE
);
```

Exercice 1.4 : Tables Sessions et Logs

📝 **Task 4:** Créez les tables pour gérer les sessions et les logs

```
CREATE TABLE sessions (
    -- À COMPLÉTER
    -- id, utilisateur_id (FK), token (unique),
    -- date_creation, date_expiration, actif
);

CREATE TABLE logs_connexion (
    -- À COMPLÉTER
    -- id, utilisateur_id (FK nullable), email_tentative,
```

```
    -- date_heure, adresse_ip, user_agent, succes, message
);
```

💡 Pourquoi `utilisateur_id` est nullable dans `logs_connexion`?

Pour logger même les tentatives avec des emails invalides!

Exercice 1.5 : Données de Test

📝 **Task 5:** Insérez des données de test

```
-- Insérer des rôles
INSERT INTO roles (nom, description) VALUES
    ('admin', 'Administrateur avec tous les droits'),
    ('moderator', 'Modérateur de contenu'),
    ('user', 'Utilisateur standard');

-- Insérer des permissions
INSERT INTO permissions (nom, ressource, action, description) VALUES
    ('read_users', 'users', 'read', 'Lire les utilisateurs'),
    ('write_users', 'users', 'write', 'Créer/modifier des utilisateurs'),
    ('delete_users', 'users', 'delete', 'Supprimer des utilisateurs'),
    ('read_posts', 'posts', 'read', 'Lire les posts'),
    ('write_posts', 'posts', 'write', 'Créer/modifier des posts'),
    ('delete_posts', 'posts', 'delete', 'Supprimer des posts');

-- À VOUS: Associez les permissions aux rôles
-- Admin: toutes les permissions
-- Moderator: read_users, read_posts, write_posts, delete_posts
-- User: read_users, read_posts, write_posts
INSERT INTO role_permissions (role_id, permission_id) VALUES
    -- À COMPLÉTER
    -- Utilisez des sous-requêtes: (SELECT id FROM roles WHERE nom = 'adm
```

Exercice 1.6 : Fonction Stockée

📝 **Task 6:** Créez une fonction pour vérifier si un utilisateur a une permission

```
CREATE OR REPLACE FUNCTION utilisateur_a_permission(
    p_utilisateur_id INT,
    p_ressource VARCHAR,
    p_action VARCHAR
)
RETURNS BOOLEAN AS $$
BEGIN
    -- À COMPLÉTER
    -- Retournez TRUE si l'utilisateur a la permission
    -- Vérifiez que l'utilisateur est actif
    -- Faites les JOINs nécessaires
END;
$$ LANGUAGE plpgsql;
```

Partie 2 : Requêtes SQL Avancées

Exercice 2.1 : Requêtes de Lecture

📝 **Task 7:** Écrivez une requête pour récupérer un utilisateur avec tous ses rôles

```
-- Utilisez array_agg pour agréger les rôles dans un tableau
SELECT
    -- À COMPLÉTER
FROM utilisateurs u
-- À COMPLÉTER (JOINs)
WHERE u.id = 1
GROUP BY u.id;
```

📝 **Task 8:** Récupérez toutes les permissions d'un utilisateur

```
SELECT DISTINCT
    u.id AS utilisateur_id,
    u.email,
    p.nom AS permission,
    p.ressource,
    p.action
FROM utilisateurs u
-- À COMPLÉTER (plusieurs JOINs nécessaires)
```

```
WHERE u.id = 1
ORDER BY p.ressource, p.action;
```

Exercice 2.2 : Statistiques et Agrégations

📝 **Task 9:** Comptez le nombre d'utilisateurs par rôle

```
SELECT
    -- À COMPLÉTER
    -- Nom du rôle et nombre d'utilisateurs
FROM roles r
LEFT JOIN -- À COMPLÉTER
GROUP BY -- À COMPLÉTER
ORDER BY nombre_utilisateurs DESC;
```

📝 **Task 10 (CHALLENGE):** Trouvez les utilisateurs qui ont le rôle 'admin' ET 'moderator'

```
-- Indice: Utilisez HAVING COUNT(DISTINCT ...) = 2
SELECT
    u.id,
    u.email,
    array_agg(r.nom) AS roles
FROM utilisateurs u
-- À COMPLÉTER
WHERE r.nom IN ('admin', 'moderator')
GROUP BY u.id, u.email
HAVING -- À COMPLÉTER
```

Exercice 2.3 : Logs et Historique

📝 **Task 11:** Comptez les tentatives de connexion échouées des 7 derniers jours

```
SELECT
    DATE(date_heure) AS jour,
    COUNT(*) AS tentatives_echouees
FROM logs_connexion
WHERE succes = false
  AND date_heure >= CURRENT_DATE - INTERVAL '7 days'
GROUP BY DATE(date_heure)
ORDER BY jour DESC;
```

Partie 3 : Backend Node.js avec Express

Exercice 3.1 : Setup du Projet

```
mkdir gestion-utilisateurs
cd gestion-utilisateurs
npm init -y

# Installer les dépendances
npm install express pg dotenv bcrypt uuid

# Créer la structure
mkdir -p database routes middleware
touch .env index.js
touch database/db.js
touch routes/authRoutes.js
touch middleware/auth.js
```

Fichier `.env` :

```
DB_USER=postgres
DB_HOST=localhost
DB_NAME=gestion_utilisateurs
DB_PASSWORD=votre_mot_de_passe
DB_PORT=5432
PORT=3000
```

Exercice 3.2 : Connexion à PostgreSQL

📝 **Task 12:** Complétez `database/db.js`

```
const { Pool } = require('pg');
require('dotenv').config();

const pool = new Pool({
    // À COMPLÉTER
    // user, host, database, password, port
});
```

```
pool.on('connect', () => {
    console.log('✅ Connecté à PostgreSQL');
});

pool.on('error', (err) => {
    console.error('❌ Erreur PostgreSQL:', err);
});

module.exports = pool;
```

Exercice 3.3 : Serveur Express

📝 **Task 13:** Créez le serveur dans `index.js`

```
const express = require('express');
const pool = require('./database/db');
const authRoutes = require('./routes/authRoutes');

const app = express();
const PORT = process.env.PORT || 3000;

// Middleware
app.use(express.json());

// Routes
app.use('/api/auth', authRoutes);

// Health check
app.get('/api/health', async (req, res) => {
    // À COMPLÉTER
    // Testez la connexion avec SELECT NOW()
});

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

Exercice 3.4 : Inscription (Register)

📝 **Task 14:** Implémentez la route POST `/api/auth/register` dans `routes/authRoutes.js`

Étapes:

1. Valider que email et password sont fournis
2. Vérifier que l'email n'existe pas déjà
3. Hasher le mot de passe avec bcrypt (salt rounds: 10)
4. Insérer l'utilisateur (BEGIN transaction)
5. Assigner le rôle "user" par défaut
6. COMMIT la transaction
7. Retourner l'utilisateur créé (sans le password_hash!)

```
const express = require('express');
const router = express.Router();
const pool = require('../database/db');
const bcrypt = require('bcrypt');

router.post('/register', async (req, res) => {
    const { email, password, nom, prenom } = req.body;

    // 1. Validation
    if (!email || !password) {
        // À COMPLÉTER
    }

    const client = await pool.connect();
    try {
        await client.query('BEGIN');

        // 2. Vérifier si email existe
        const checkUser = await client.query(
            // À COMPLÉTER
        );

        if (checkUser.rows.length > 0) {
            // À COMPLÉTER
        }

        // 3. Hasher le mot de passe
        const passwordHash = await bcrypt.hash(password, 10);

        // 4. Insérer l'utilisateur
        const result = await client.query(
            // À COMPLÉTER
            // RETURNING id, email, nom, prenom, date_creation
        );
```

```
        const newUser = result.rows[0];

        // 5. Assigner le rôle "user"
        await client.query(
            // À COMPLÉTER
            // Sous-requête: SELECT id FROM roles WHERE nom = 'user'
        );

        await client.query('COMMIT');

        res.status(201).json({
            message: 'Utilisateur créé avec succès',
            user: newUser
        });

    } catch (error) {
        await client.query('ROLLBACK');
        console.error('Erreur création utilisateur:', error);
        res.status(500).json({ error: 'Erreur serveur' });
    } finally {
        client.release();
    }
});

module.exports = router;
```

Exercice 3.5 : Connexion (Login)

📝 **Task 15 (CHALLENGE):** Implémentez POST `/api/auth/login`

Étapes:

1. Récupérer l'utilisateur par email
2. Vérifier que l'utilisateur existe et est actif
3. Comparer le mot de passe avec bcrypt.compare()
4. Générer un token (UUID v4)
5. Créer une session (expiration: 24h)
6. Logger la tentative de connexion (réussie ou échouée)
7. Retourner le token et les infos utilisateur

```
const { v4: uuidv4 } = require('uuid');

router.post('/login', async (req, res) => {
    const { email, password } = req.body;

    const client = await pool.connect();
    try {
        await client.query('BEGIN');

        // 1. Récupérer l'utilisateur
        const userResult = await client.query(
            // À COMPLÉTER
        );

        if (userResult.rows.length === 0) {
            // Logger l'échec
            await client.query(
                // À COMPLÉTER: INSERT INTO logs_connexion
            );
            await client.query('COMMIT');
            return res.status(401).json({ error: 'Email ou mot de passe i
        }

        const user = userResult.rows[0];

        // 2. Vérifier si actif
        if (!user.actif) {
            // À COMPLÉTER
        }

        // 3. Vérifier le mot de passe
        const passwordMatch = await bcrypt.compare(password, user.passwor

        if (!passwordMatch) {
            // À COMPLÉTER
        }

        // 4. Générer token
        const token = uuidv4();
        const expiresAt = new Date();
        expiresAt.setHours(expiresAt.getHours() + 24);

        // 5. Créer session
        await client.query(
```

```
            // À COMPLÉTER
        );

        // 6. Logger succès
        await client.query(
            // À COMPLÉTER
        );

        await client.query('COMMIT');

        res.json({
            message: 'Connexion réussie',
            token: token,
            user: {
                id: user.id,
                email: user.email,
                nom: user.nom,
                prenom: user.prenom
            },
            expiresAt: expiresAt
        });

    } catch (error) {
        await client.query('ROLLBACK');
        console.error('Erreur login:', error);
        res.status(500).json({ error: 'Erreur serveur' });
    } finally {
        client.release();
    }
});
```

Exercice 3.6 : Middleware d'Authentification

📝 **Task 16:** Créez le middleware dans `middleware/auth.js`

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

async function requireAuth(req, res, next) {
    const token = req.headers['authorization'];

    if (!token) {
        // À COMPLÉTER
```

```
    }

    try {
        // Vérifier que le token est valide
        const result = await pool.query(
            // À COMPLÉTER
            // JOIN avec utilisateurs
            // Vérifier: actif, date_expiration, session active
        );

        if (result.rows.length === 0) {
            // À COMPLÉTER
        }

        req.user = result.rows[0];
        next();

    } catch (error) {
        console.error('Erreur middleware auth:', error);
        res.status(500).json({ error: 'Erreur serveur' });
    }
}

module.exports = { requireAuth };
```

Exercice 3.7 : Route Protégée - Profil

📝 **Task 17:** Ajoutez une route pour récupérer le profil de l'utilisateur connecté

```
const { requireAuth } = require('../middleware/auth');

// GET /api/auth/profile
router.get('/profile', requireAuth, async (req, res) => {
    try {
        // Récupérer l'utilisateur avec ses rôles
        const result = await pool.query(
            // À COMPLÉTER
            // Utilisez array_agg pour les rôles
        );

        res.json({ user: result.rows[0] });
```

```
    } catch (error) {
        console.error('Erreur profil:', error);
        res.status(500).json({ error: 'Erreur serveur' });
    }
});
```

Exercice 3.8 : CHALLENGE - Middleware de Permissions

📝 **Task 18 (BONUS):** Créez un middleware pour vérifier les permissions

```
// Dans middleware/auth.js
function requirePermission(ressource, action) {
    return async (req, res, next) => {
        try {
            const result = await pool.query(
                'SELECT utilisateur_a_permission($1, $2, $3) AS a_permiss
                [req.user.utilisateur_id, ressource, action]
            );

            if (!result.rows[0].a_permission) {
                return res.status(403).json({ error: 'Permission refusée'
            }

            next();

        } catch (error) {
            console.error('Erreur vérification permission:', error);
            res.status(500).json({ error: 'Erreur serveur' });
        }
    };
}

module.exports = { requireAuth, requirePermission };
```

Utilisation:

```
router.delete('/users/:id',
    requireAuth,
    requirePermission('users', 'delete'),
    async (req, res) => {
        // Supprimer l'utilisateur
```

```
    }
);
```

Exercice 3.9 : CRUD Complet - Lire les Utilisateurs

📝 **Task 19:** Implémentez GET `/api/users` pour lister les utilisateurs avec pagination

```
// Dans routes/userRoutes.js
const express = require('express');
const router = express.Router();
const pool = require('../database/db');
const { requireAuth, requirePermission } = require('../middleware/auth');

// GET /api/users?page=1&limit=10
router.get('/',
    requireAuth,
    requirePermission('users', 'read'),
    async (req, res) => {
        const page = parseInt(req.query.page) || 1;
        const limit = parseInt(req.query.limit) || 10;
        const offset = (page - 1) * limit;

        try {
            // À COMPLÉTER
            // 1. Compter le total d'utilisateurs
            // 2. Récupérer les utilisateurs avec leurs rôles (array_agg)
            // 3. Utiliser LIMIT et OFFSET pour la pagination
            // 4. Retourner users et pagination info

        } catch (error) {
            console.error('Erreur liste utilisateurs:', error);
            res.status(500).json({ error: 'Erreur serveur' });
        }
    }
);

module.exports = router;
```

Exercice 3.10 : CRUD - Mettre à Jour un Utilisateur

📝 **Task 20:** Implémentez PUT `/api/users/:id` pour modifier un utilisateur

```
// PUT /api/users/:id
router.put('/:id',
    requireAuth,
    requirePermission('users', 'write'),
    async (req, res) => {
        const { id } = req.params;
        const { nom, prenom, actif } = req.body;

        try {
            const result = await pool.query(
                // À COMPLÉTER
                // UPDATE utilisateurs
                // SET nom = $1, prenom = $2, actif = $3, date_modificati
                // WHERE id = $4
                // RETURNING id, email, nom, prenom, actif, date_modifica
            );

            if (result.rows.length === 0) {
                return res.status(404).json({ error: 'Utilisateur non trc
            }

            res.json({
                message: 'Utilisateur mis à jour',
                user: result.rows[0]
            });

        } catch (error) {
            console.error('Erreur mise à jour utilisateur:', error);
            res.status(500).json({ error: 'Erreur serveur' });
        }
    }
);
```

Exercice 3.11 : CRUD - Supprimer un Utilisateur

📝 **Task 21:** Implémentez DELETE `/api/users/:id` pour supprimer un utilisateur

⚠️ **Important:** Un utilisateur ne peut pas se supprimer lui-même!

```
// DELETE /api/users/:id
router.delete('/:id',
    requireAuth,
```

```
    requirePermission('users', 'delete'),
    async (req, res) => {
        const { id } = req.params;

        // Empêcher l'auto-suppression
        if (parseInt(id) === req.user.utilisateur_id) {
            return res.status(400).json({
                error: 'Vous ne pouvez pas supprimer votre propre compte'
            });
        }

        try {
            const result = await pool.query(
                // À COMPLÉTER
                // DELETE FROM utilisateurs WHERE id = $1 RETURNING id, e
            );

            if (result.rows.length === 0) {
                return res.status(404).json({ error: 'Utilisateur non trc
            }

            res.json({
                message: 'Utilisateur supprimé',
                user: result.rows[0]
            });

        } catch (error) {
            console.error('Erreur suppression utilisateur:', error);
            res.status(500).json({ error: 'Erreur serveur' });
        }
    }
);
```

Exercice 3.12 : Utiliser la Fonction Stockée

📝 **Task 22:** Créez une route pour vérifier les permissions d'un utilisateur

```
// GET /api/users/:id/permissions
router.get('/:id/permissions',
    requireAuth,
    async (req, res) => {
        const { id } = req.params;
```

```
        try {
            // Récupérer toutes les permissions de l'utilisateur
            const result = await pool.query(
                // À COMPLÉTER
                // SELECT DISTINCT p.nom, p.ressource, p.action, p.descri
                // FROM utilisateurs u
                // INNER JOIN utilisateur_roles ...
                // WHERE u.id = $1
            );

            res.json({
                utilisateur_id: parseInt(id),
                permissions: result.rows
            });

        } catch (error) {
            console.error('Erreur récupération permissions:', error);
            res.status(500).json({ error: 'Erreur serveur' });
        }
    }
);
```

Exercice 3.13 : Fonction est_token_valide()

📝 **Task 23:** Ajoutez une fonction stockée pour valider les tokens

D'abord, créez la fonction dans votre base de données:

```
CREATE OR REPLACE FUNCTION est_token_valide(p_token VARCHAR)
RETURNS BOOLEAN AS $$
BEGIN
    RETURN EXISTS (
        SELECT 1
        FROM sessions s
        INNER JOIN utilisateurs u ON s.utilisateur_id = u.id
        WHERE s.token = p_token
          AND s.actif = true
          AND s.date_expiration > CURRENT_TIMESTAMP
          AND u.actif = true
    );
END;
$$ LANGUAGE plpgsql;
```

Puis, utilisez-la dans votre middleware (optionnel, amélioration):

```
// Alternative au middleware requireAuth
async function requireAuthWithFunction(req, res, next) {
    const token = req.headers['authorization'];

    if (!token) {
        return res.status(401).json({ error: 'Token manquant' });
    }

    try {
        // Utiliser la fonction stockée
        const validResult = await pool.query(
            'SELECT est_token_valide($1) AS valide',
            [token]
        );

        if (!validResult.rows[0].valide) {
            return res.status(401).json({ error: 'Token invalide ou expir
        }

        // Récupérer les infos utilisateur
        const userResult = await pool.query(
            `SELECT s.utilisateur_id, u.email, u.nom, u.prenom
             FROM sessions s
             INNER JOIN utilisateurs u ON s.utilisateur_id = u.id
             WHERE s.token = $1`,
            [token]
        );

        req.user = userResult.rows[0];
        next();

    } catch (error) {
        console.error('Erreur middleware auth:', error);
        res.status(500).json({ error: 'Erreur serveur' });
    }
}
```

Exercice 3.14 : Déconnexion (Logout)

📝 **Task 24:** Implémentez POST `/api/auth/logout`

```
// POST /api/auth/logout
router.post('/logout', requireAuth, async (req, res) => {
    const token = req.headers['authorization'];

    try {
        // À COMPLÉTER
        // 1. Désactiver la session
        // 2. Logger la déconnexion dans logs_connexion
    } catch (error) {
        console.error('Erreur logout:', error);
        res.status(500).json({ error: 'Erreur serveur' });
    }
});
```

Exercice 3.15 : Logs de Connexion

📝 **Task 25:** Récupérez l'historique des connexions d'un utilisateur

```
// GET /api/auth/logs
router.get('/logs', requireAuth, async (req, res) => {
    try {
        const result = await pool.query(
            // À COMPLÉTER
            // SELECT * FROM logs_connexion
            // WHERE utilisateur_id = $1
            // ORDER BY date_heure DESC
            // LIMIT 50
        );

        res.json({ logs: result.rows });

    } catch (error) {
        console.error('Erreur logs:', error);
        res.status(500).json({ error: 'Erreur serveur' });
    }
});
```

N'oubliez pas: Dans `index.js`, montez les routes utilisateurs:

```
const userRoutes = require('./routes/userRoutes');
app.use('/api/users', userRoutes);
```

Tests avec Postman / cURL

1. Inscription

```
curl -X POST http://localhost:3000/api/auth/register \
  -H "Content-Type: application/json" \
  -d '{
    "email": "alice@example.com",
    "password": "password123",
    "nom": "Dupont",
    "prenom": "Alice"
  }'
```

2. Connexion

```
curl -X POST http://localhost:3000/api/auth/login \
  -H "Content-Type: application/json" \
  -d '{
    "email": "alice@example.com",
    "password": "password123"
  }'
```

Réponse: Copiez le `token` retourné

3. Profil (avec token)

```
curl -X GET http://localhost:3000/api/auth/profile \
  -H "Authorization: VOTRE_TOKEN_ICI"
```

Livrables Attendus

À la fin du TP, vous devez avoir:

Base de données:

- ✅ Base de données complète avec 7 tables
- ✅ Fonction stockée `utilisateur_a_permission()`
- ✅ Fonction stockée `est_token_valide()` (optionnel)

API REST complète:

- ✅ Route POST `/api/auth/register` - Création utilisateur avec transactions
- ✅ Route POST `/api/auth/login` - Authentification avec token UUID
- ✅ Route POST `/api/auth/logout` - Déconnexion
- ✅ Route GET `/api/auth/profile` - Profil utilisateur
- ✅ Route GET `/api/auth/logs` - Historique des connexions

CRUD Utilisateurs:

- ✅ Route GET `/api/users` - Liste avec pagination
- ✅ Route GET `/api/users/:id` - Détails d'un utilisateur
- ✅ Route PUT `/api/users/:id` - Mise à jour
- ✅ Route DELETE `/api/users/:id` - Suppression (avec protection auto-suppression)
- ✅ Route GET `/api/users/:id/permissions` - Permissions d'un utilisateur

Middleware et Sécurité:

- ✅ Middleware d'authentification (`requireAuth`)
- ✅ Middleware de permissions (`requirePermission`)
- ✅ Logs de connexion (réussies et échouées)
- ✅ Hachage des mots de passe avec bcrypt
- ✅ Gestion des erreurs et transactions SQL

💡 Pour Aller Plus Loin : Architecture RCS (Router-Controller-Service)

Observation Importante

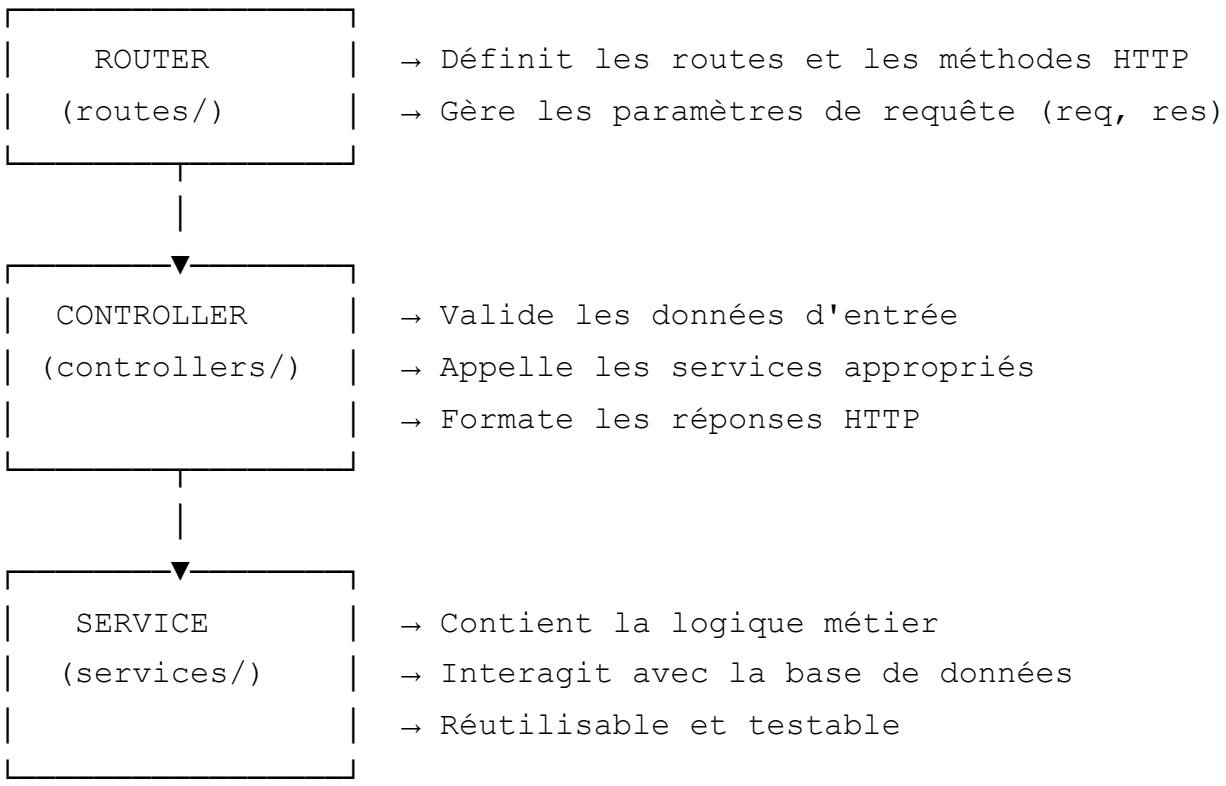
Dans ce TP, nous avons placé toute la logique métier directement dans les **routes**. C'est une approche simple et fonctionnelle pour débuter, mais **ce n'est pas la meilleure pratique** pour des applications réelles.

Problèmes de l'approche actuelle :

- ❌ Routes trop volumineuses et difficiles à maintenir
- ❌ Logique métier mélangée avec la logique de routage
- ❌ Code difficile à tester unitairement
- ❌ Duplication de code entre différentes routes
- ❌ Impossible de réutiliser la logique dans d'autres contextes

L'Architecture RCS

Vous avez appris l'approche **RCS (Router-Controller-Service)** en cours (voir `week_3b`). Cette architecture sépare les responsabilités en 3 couches :



🎯 Exercice Bonus : Refactorisation en Architecture RCS

Objectif : Refactorisez votre code pour suivre l'architecture RCS.

Étape 1 : Créer la structure de dossiers

```
mkdir controllers services
touch controllers/authController.js
touch controllers/userController.js
touch services/authService.js
touch services/userService.js
```

Étape 2 : Exemple de refactorisation - Route Register

Avant (tout dans la route) :

```
// routes/authRoutes.js
router.post('/register', async (req, res) => {
    const { email, password, nom, prenom } = req.body;

    if (!email || !password) {
        return res.status(400).json({ error: 'Email et mot de passe requi
    }

    const client = await pool.connect();
    try {
        await client.query('BEGIN');
        const checkUser = await client.query(
            'SELECT id FROM utilisateurs WHERE email = $1',
            [email]
        );
        // ... 50 lignes de logique métier ...
    } catch (error) {
        // ...
    }
});
```

Après (Architecture RCS) :

```
// routes/authRoutes.js
const authController = require('../controllers/authController');

router.post('/register', authController.register);
router.post('/login', authController.login);
router.post('/logout', requireAuth, authController.logout);
router.get('/profile', requireAuth, authController.getProfile);


// controllers/authController.js
const authService = require('../services/authService');

exports.register = async (req, res) => {
    try {
        const { email, password, nom, prenom } = req.body;

        // Validation
        if (!email || !password) {
```

```
            return res.status(400).json({
                error: 'Email et mot de passe requis'
            });
        }

        // Appel au service
        const user = await authService.registerUser({
            email,
            password,
            nom,
            prenom
        });

        res.status(201).json({
            message: 'Utilisateur créé avec succès',
            user
        });

    } catch (error) {
        if (error.code === 'EMAIL_EXISTS') {
            return res.status(409).json({ error: error.message });
        }
        console.error('Erreur création utilisateur:', error);
        res.status(500).json({ error: 'Erreur serveur' });
    }
};

exports.login = async (req, res) => {
    try {
        const { email, password } = req.body;

        if (!email || !password) {
            return res.status(400).json({
                error: 'Email et mot de passe requis'
            });
        }

        const result = await authService.login(email, password);

        res.json(result);

    } catch (error) {
        if (error.code === 'INVALID_CREDENTIALS') {
            return res.status(401).json({ error: error.message });
```

```
        }
        if (error.code === 'ACCOUNT_INACTIVE') {
            return res.status(403).json({ error: error.message });
        }
        console.error('Erreur login:', error);
        res.status(500).json({ error: 'Erreur serveur' });
    }
};


// services/authService.js
const pool = require('../database/db');
const bcrypt = require('bcrypt');
const { v4: uuidv4 } = require('uuid');

exports.registerUser = async ({ email, password, nom, prenom }) => {
    const client = await pool.connect();

    try {
        await client.query('BEGIN');

        // Vérifier si l'email existe
        const checkUser = await client.query(
            'SELECT id FROM utilisateurs WHERE email = $1',
            [email]
        );

        if (checkUser.rows.length > 0) {
            await client.query('ROLLBACK');
            const error = new Error('Email déjà utilisé');
            error.code = 'EMAIL_EXISTS';
            throw error;
        }

        // Hasher le mot de passe
        const passwordHash = await bcrypt.hash(password, 10);

        // Insérer l'utilisateur
        const result = await client.query(
            `INSERT INTO utilisateurs (email, password_hash, nom, prenom)
             VALUES ($1, $2, $3, $4)
             RETURNING id, email, nom, prenom, date_creation`,
            [email, passwordHash, nom, prenom]
        );
```

```
        const newUser = result.rows[0];

        // Assigner le rôle "user"
        await client.query(
            `INSERT INTO utilisateur_roles (utilisateur_id, role_id)
             SELECT $1, id FROM roles WHERE nom = 'user'`,
            [newUser.id]
        );

        await client.query('COMMIT');

        return newUser;

    } catch (error) {
        await client.query('ROLLBACK');
        throw error;
    } finally {
        client.release();
    }
};

exports.login = async (email, password) => {
    const client = await pool.connect();

    try {
        await client.query('BEGIN');

        // Récupérer l'utilisateur
        const userResult = await client.query(
            `SELECT id, email, password_hash, nom, prenom, actif
             FROM utilisateurs WHERE email = $1`,
            [email]
        );

        if (userResult.rows.length === 0) {
            await logFailedLogin(client, email, null, 'Email inexistant')
            await client.query('COMMIT');

            const error = new Error('Email ou mot de passe incorrect');
            error.code = 'INVALID_CREDENTIALS';
            throw error;
        }

        const user = userResult.rows[0];
```

```
    // Vérifier si actif
    if (!user.actif) {
        await logFailedLogin(client, email, user.id, 'Compte désactiv
        await client.query('COMMIT');

        const error = new Error('Compte désactivé');
        error.code = 'ACCOUNT_INACTIVE';
        throw error;
    }

    // Vérifier le mot de passe
    const passwordMatch = await bcrypt.compare(password, user.passwor

    if (!passwordMatch) {
        await logFailedLogin(client, email, user.id, 'Mot de passe ir
        await client.query('COMMIT');

        const error = new Error('Email ou mot de passe incorrect');
        error.code = 'INVALID_CREDENTIALS';
        throw error;
    }

    // Générer token
    const token = uuidv4();
    const expiresAt = new Date();
    expiresAt.setHours(expiresAt.getHours() + 24);

    // Créer session
    await client.query(
        `INSERT INTO sessions (utilisateur_id, token, date_expiratior
         VALUES ($1, $2, $3)`,
        [user.id, token, expiresAt]
    );

    // Logger succès
    await client.query(
        `INSERT INTO logs_connexion
         (utilisateur_id, email_tentative, succes, message)
         VALUES ($1, $2, true, 'Connexion réussie')`,
        [user.id, email]
    );

    await client.query('COMMIT');
```

```
        return {
            message: 'Connexion réussie',
            token,
            user: {
                id: user.id,
                email: user.email,
                nom: user.nom,
                prenom: user.prenom
            },
            expiresAt
        };

    } catch (error) {
        await client.query('ROLLBACK');
        throw error;
    } finally {
        client.release();
    }
};

// Fonction helper privée
async function logFailedLogin(client, email, userId, message) {
    await client.query(
        `INSERT INTO logs_connexion
         (utilisateur_id, email_tentative, succes, message)
         VALUES ($1, $2, false, $3)`,
        [userId, email, message]
    );
}
```

Avantages de l'Architecture RCS

✅ **Séparation des responsabilités** : Chaque couche a un rôle clair
✅ **Testabilité** : Les services peuvent être testés indépendamment
✅ **Réutilisabilité** : La logique métier peut être utilisée ailleurs (CLI, workers, etc.)
✅ **Maintenabilité** : Code plus lisible et organisé
✅ **Évolutivité** : Facile d'ajouter de nouvelles fonctionnalités
✅ **Gestion d'erreurs** : Erreurs personnalisées avec codes d'erreur

🎯 Votre Mission

Refactorisez **tout votre code** pour suivre l'architecture RCS :

1. **authService.js** : `registerUser()`, `login()`, `logout()`, `getUserProfile()`, `getUserLogs()`
2. **userService.js** : `getAllUsers()`, `getUserById()`, `updateUser()`, `deleteUser()`, `getUserPermissions()`
3. **authController.js** : Tous les contrôleurs d'authentification
4. **userController.js** : Tous les contrôleurs de gestion utilisateurs

Critères de validation :

- ✅ Aucune logique métier dans les routes
- ✅ Les contrôleurs ne font que valider et appeler les services
- ✅ Tous les appels à la base de données sont dans les services
- ✅ Gestion d'erreurs avec codes d'erreur personnalisés
- ✅ Code réutilisable et testable

Bon courage! 🚀