

TP - Migration SQL vers MongoDB avec Cache Redis

Système d'Authentification avec Rôles et Permissions

BUT S5 - IUT Nord Franche-Comté

Durée: 1h30

Objectif: Migrer une base de données relationnelle vers MongoDB et implémenter un système de cache Redis

Contexte

Vous travaillez sur un système d'authentification existant qui utilise actuellement PostgreSQL. Votre mission est de migrer cette base vers MongoDB et d'optimiser les performances avec Redis.

Base de données SQL existante (fictive)

Voici le schéma SQL actuel que vous devez migrer :

```
-- Table des utilisateurs
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  username VARCHAR(100) UNIQUE NOT NULL,
  email VARCHAR(255) UNIQUE NOT NULL,
  password_hash VARCHAR(255) NOT NULL,
  is_active BOOLEAN DEFAULT true,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Table des rôles
CREATE TABLE roles (
  id SERIAL PRIMARY KEY,
  name VARCHAR(50) UNIQUE NOT NULL,
  description TEXT
);

-- Table des permissions
CREATE TABLE permissions (
  id SERIAL PRIMARY KEY,
  name VARCHAR(100) UNIQUE NOT NULL,
  resource VARCHAR(100) NOT NULL,
  action VARCHAR(50) NOT NULL
);

-- Table d'association users-roles (Many-to-Many)
CREATE TABLE user_roles (
  user_id INTEGER REFERENCES users(id),
  role_id INTEGER REFERENCES roles(id),
  assigned_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (user_id, role_id)
);

-- Table d'association roles-permissions (Many-to-Many)
CREATE TABLE role_permissions (
  role_id INTEGER REFERENCES roles(id),
  permission_id INTEGER REFERENCES permissions(id),
  PRIMARY KEY (role_id, permission_id)
);
```

Exemples de données dans les tables SQL

Table `users`

id	username	email	password_hash	is_active	created_at
1	alice	alice@example.com	\$2b\$10\$xYz...	true	2024-01-15 10:30:00
2	bob	bob@example.com	\$2b\$10\$aBc...	true	2024-01-16 14:20:00
3	charlie	charlie@example.com	\$2b\$10\$dEf...	false	2024-01-17 09:15:00

Table `roles`

id	name	description
1	admin	Administrateur avec tous les droits
2	moderator	Modérateur du contenu
3	user	Utilisateur standard

Table `permissions`

id	name	resource	action
1	read_users	users	read
2	write_users	users	write
3	delete_users	users	delete
4	read_posts	posts	read
5	write_posts	posts	write

Table `user_roles` (association)

user_id	role_id	assigned_at
1	1	2024-01-15 10:30:00
2	2	2024-01-16 14:20:00
2	3	2024-01-16 14:20:00
3	3	2024-01-17 09:15:00

Table `role_permissions` (association)

role_id	permission_id
1	1
1	2
1	3
1	4
1	5
2	1
2	4
2	5

role_id	permission_id
3	4

🔔 **Observations importantes:**

- Un utilisateur peut avoir **plusieurs rôles** (bob est à la fois moderator ET user)
- Un rôle a **plusieurs permissions** (admin a toutes les permissions)
- En SQL, ces relations Many-to-Many nécessitent des **tables de jointure**
- En MongoDB, vous pouvez **embarquer** ces données ou utiliser des **références**

Partie 1 : Modélisation MongoDB (20 minutes)

Exercice 1.1 : Créer les schémas Mongoose

Créez un dossier `tp-auth-system` avec la structure suivante :

```
tp-auth-system/  
├── models/  
│   ├── User.js  
│   ├── Role.js  
│   └── Permission.js  
├── routes/  
│   └── authRoutes.js  
├── services/  
│   └── cache.js  
└── index.js
```

Question 1: Dans `models/User.js`, créez le schéma Mongoose pour les utilisateurs.

🔔 **Indice:** Contrairement à SQL, MongoDB permet d'**embarquer** (embed) les relations. Réfléchissez si vous voulez :

- Référencer les rôles par leur ID (comme en SQL avec foreign keys)
- OU embarquer directement les rôles dans le document utilisateur

Exemple de structure attendue:

```
const mongoose = require('mongoose');  
const { Schema } = mongoose;  
  
const userSchema = new Schema({  
  username: { type: String, required: true, unique: true },  
  // TODO: Complétez les autres champs  
  // TODO: Décidez comment représenter la relation avec roles  
});  
  
module.exports = mongoose.model('User', userSchema);
```

Question 2: Créez les schémas pour `Role.js` et `Permission.js`.

Partie 2 : API REST avec Express (30 minutes)

Exercice 2.1 : Configuration de base

Dans `index.js`, configurez Express et la connexion MongoDB :

```
const express = require('express');
const mongoose = require('mongoose');
const bodyParser = require('body-parser');

mongoose.connect('mongodb://localhost:27017/tp_auth_system', {
  useNewUrlParser: true,
  useUnifiedTopology: true
});

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

// TODO: Importer et configurer les routes

const PORT = 5000; // ou 3000
app.listen(PORT, () => {
  console.log(`Serveur démarré sur le port ${PORT}`);
});
```

Exercice 2.2 : Routes CRUD de base

Dans `routes/authRoutes.js`, implémentez les routes suivantes :

Route 1 : GET `/api/users`- Lire tous les utilisateurs

```
// TODO: Récupérer tous les utilisateurs
// Pensez à utiliser .populate() si vous avez utilisé des références
```

Route 2 : GET `/api/users/:id`- Lire un utilisateur par ID

```
// TODO: Récupérer un utilisateur spécifique
```

Route 3 : POST `/api/users`- Créer un utilisateur

```
// Body attendu:
// {
//   "username": "john_doe",
//   "email": "john@example.com",
//   "password_hash": "hashed_password",
//   "roles": [...]
// }
```

Route 4 : DELETE `/api/users/:id`- Supprimer un utilisateur

```
// TODO: Supprimer un utilisateur
```

Partie 3 : Agrégations MongoDB (20 minutes)

Les agrégations remplacent les JOINS SQL. Implémentez deux routes d'agrégation :

Exercice 3.1 : GET `/api/users/:id/with-permissions`

Créez une route qui retourne un utilisateur avec **tous ses rôles ET toutes ses permissions**.

📌 **Rappel du TD:** Utilisez `$lookup` pour faire des jointures entre collections.

Exemple de résultat attendu:

```
{
  "_id": "507f1f77bcf86cd799439011",
  "username": "john_doe",
  "email": "john@example.com",
  "roles": [
    {
      "name": "admin",
      "permissions": [
        { "name": "read_users", "resource": "users", "action": "read" },
        { "name": "write_users", "resource": "users", "action": "write" }
      ]
    }
  ]
}
```

Structure de pipeline suggérée:

```
app.get('/api/users/:id/with-permissions', async (req, res) => {
  try {
    const result = await User.aggregate([
      { $match: { _id: mongoose.Types.ObjectId(req.params.id) } },
      // TODO: $lookup pour joindre avec la collection roles
      // TODO: $lookup imbriqué pour joindre avec permissions
      // TODO: $project pour formater le résultat
    ]);
    res.json(result[0]);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});
```

Exercice 3.2 : GET /api/stats/users-by-role

Créez une route qui compte le nombre d'utilisateurs par rôle.

Résultat attendu:

```
[
  { "_id": "admin", "count": 5 },
  { "_id": "user", "count": 120 },
  { "_id": "moderator", "count": 12 }
]
```

🔗 **Indice:** Utilisez `$group` avec `$sum`

Partie 4 : Cache Redis (20 minutes)

Exercice 4.1 : Configuration du cache

Dans `services/cache.js`, créez un middleware de cache réutilisable (inspiré du TD) :

```

const redis = require('redis');
const redisClient = redis.createClient({ url: 'redis://127.0.0.1:6379' });

let connected = false;

async function connect() {
  try {
    await redisClient.connect();
    connected = true;
    console.log('☑ Connected to Redis');
  } catch (error) {
    console.error('☹ Redis error:', error.message);
  }
}

connect();

function cacheMiddleware(keyPrefix, ttl = 60) {
  return async (req, res, next) => {
    // TODO: Implémenter la logique de cache
    // 1. Créer une clé unique basée sur l'URL
    // 2. Vérifier si la donnée existe dans Redis
    // 3. Si OUI: retourner depuis le cache (CACHE HIT)
    // 4. Si NON: passer au next(), et intercepter res.json pour mettre en cache
  };
}

// TODO: Fonction pour invalider le cache
async function invalidateCache(pattern) {
  // ...
}

module.exports = { cacheMiddleware, invalidateCache };

```

Exercice 4.2 : Appliquer le cache

Question 1: Ajoutez le cache aux routes GET :

```

const { cacheMiddleware } = require('../services/cache');

// Cache de 60 secondes pour un utilisateur
app.get('/api/users/:id', cacheMiddleware('user', 60), async (req, res) => {
  // ... votre code existant
});

// Cache de 300 secondes (5 minutes) pour la liste
app.get('/api/users', cacheMiddleware('users', 300), async (req, res) => {
  // ... votre code existant
});

```

Question 2: Implémentez le **Write-Through** pour POST et DELETE :

```
const { invalidateCache } = require('../services/cache');

app.post('/api/users', async (req, res) => {
  // TODO:
  // 1. Créer l'utilisateur en base
  // 2. Invalider le cache de la liste des utilisateurs
});

app.delete('/api/users/:id', async (req, res) => {
  // TODO:
  // 1. Supprimer l'utilisateur de la base
  // 2. Invalider le cache de cet utilisateur
  // 3. Invalider le cache de la liste
});
```

Tests et Validation

Données de test

Créez quelques données de test via Postman ou curl :

```
# Créer une permission
POST http://localhost:5000/api/permissions
{
  "name": "read_users",
  "resource": "users",
  "action": "read"
}

# Créer un rôle
POST http://localhost:5000/api/roles
{
  "name": "admin",
  "description": "Administrateur système",
  "permissions": ["<id_permission>"]
}

# Créer un utilisateur
POST http://localhost:5000/api/users
{
  "username": "alice",
  "email": "alice@example.com",
  "password_hash": "hashed_pwd_123",
  "roles": ["<id_role>"]
}
```

Vérification du cache

1. **Première requête** : GET http://localhost:5000/api/users
 - Doit afficher `⚠ CACHE MISS` dans la console
 - Temps de réponse: ~50-100ms
 2. **Deuxième requête** : GET http://localhost:5000/api/users
 - Doit afficher `⚠ CACHE HIT` dans la console
 - Temps de réponse: ~5-10ms
 3. **Après POST/DELETE** :
 - Le cache doit être invalidé
 - La prochaine requête GET doit être un CACHE MISS
-

Livrables

À la fin du TP, vous devez avoir :

- 3 modèles Mongoose (User, Role, Permission)
 - 4 routes CRUD (GET all, GET by id, POST, DELETE)
 - 2 routes d'agrégation (with-permissions, stats)
 - Middleware de cache Redis fonctionnel
 - Stratégie Write-Through implémentée
-

Questions de Réflexion

1. Pourquoi utiliser MongoDB plutôt que PostgreSQL pour ce cas d'usage ?
 - Réponse attendue: Flexibilité du schéma, embedded documents, performance sur les lectures
 2. Quelle est la différence entre Cache-Aside et Write-Through ?
 - Cache-Aside (Lazy Loading): On charge en cache uniquement quand on lit
 - Write-Through: On met à jour le cache immédiatement lors des écritures
 3. Que se passe-t-il si Redis tombe en panne ?
 - L'application doit continuer à fonctionner (mode dégradé sans cache)
-

Pour aller plus loin (Bonus)

- Implémentez le hachage de mot de passe avec bcrypt
- Ajoutez un système de TTL différent par rôle (admin: 1h, user: 5min)
- Utilisez Redis HSET/HGET pour stocker les users comme des Hash plutôt que des strings JSON

Bon courage ! 🍀