Cours MERN - Projet Avancé de Mi-Formation

API pour une Plateforme de Cours en Ligne : Maîtrise des Relations de Données

Abdelweheb GUEDDES & Mohamed Ben Jazia / Ecole Polytechnique Sousse

9 septembre 2025

Table des matières

1 Introduction et Objectifs

Ce projet avancé est conçu pour vous faire maîtriser la modélisation et l'implémentation des relations de données, un aspect fondamental de la plupart des applications complexes. Vous allez construire une API REST pour une mini-plateforme de cours en ligne, en mettant en œuvre les trois types de relations : One-to-One, One-to-Many, et Many-to-Many.

1.1 Compétences à Démontrer

- Implémentation d'un CRUD complet pour plusieurs ressources.
- Modélisation et implémentation d'une relation **One-to-One** (Utilisateur ↔ Profil).
- Modélisation et implémentation d'une relation **One-to-Many** (Cours → Critiques).
- Modélisation et implémentation d'une relation **Many-to-Many** (Utilisateurs ↔ Cours).
- Utilisation avancée de Mongoose pour gérer les relations (`ref`, `populate`).
- Conception d'une structure de routes logique pour les ressources imbriquées.

Cahier des Charges

1.2 Concept Général

L'API "EduPlatform" gérera des utilisateurs, leurs profils, des cours, des inscriptions à ces cours, et des critiques sur les cours.

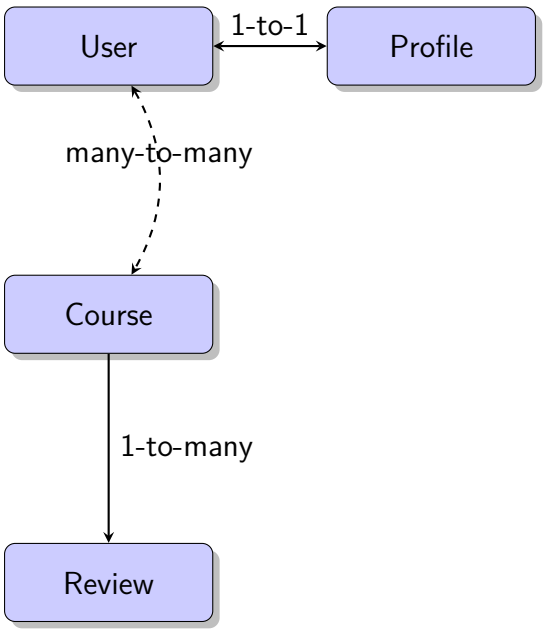

FIGURE 1 – Schéma des relations entre les modèles.

1.3 Modèles de Données

Vous devrez créer quatre modèles Mongoose.

1.3.1 Modèle `User`

- `username` : String, requis, unique.
- `email` : String, requis, unique, format email valide.
- `courses` : [**Many-to-Many**] Un tableau d'ObjectIDs faisant référence au modèle 'Course'.
  - `type: [ type: mongoose.Schema.Types.ObjectId, ref: 'Course' ]`

1.3.2 Modèle `Profile`

- `user` : [**One-to-One**] Une référence ObjectID à un 'User'. Requis, unique.
- `bio` : String.
- `website` : String.

1.3.3 Modèle `Course`

- `title` : String, requis, unique.
- `description` : String, requis.
- `instructor` : String, requis.
- `students` : [**Many-to-Many**] Un tableau d'ObjectIDs faisant référence au modèle 'User'.

1.3.4 Modèle `Review`

- `rating` : Number, requis, doit être entre 1 et 5.
- `comment` : String.
- `course` : [**One-to-Many**] Une référence ObjectID à un 'Course'. Requis.
- `user` : Une référence ObjectID à un 'User'. Requis.

1.4 Fonctionnalités Requises (Points de Terminaison)

1.4.1 Gestion des Utilisateurs

- `POST /api/users` : Crée un utilisateur.
- `GET /api/users` : Récupère tous les utilisateurs.
- `GET /api/users/:id` : Récupère un utilisateur.

1.4.2 Gestion des Profils (Relation 1-to-1)

- `POST /api/users/:userId/profile` : Crée un profil pour un utilisateur.
- `GET /api/users/:userId/profile` : Récupère le profil d'un utilisateur.
- `PUT /api/users/:userId/profile` : Met à jour le profil d'un utilisateur.

1.4.3 Gestion des Cours

- `POST /api/courses` : Crée un cours.
- `GET /api/courses` : Récupère tous les cours.
- `GET /api/courses/:id` : Récupère un cours.

1.4.4 Gestion des Inscriptions (Relation Many-to-Many)

— `POST /api/courses/:courseId/enroll` : Inscrit un utilisateur (dont l'ID sera dans le body) à un cours. Doit mettre à jour à la fois le tableau 'students' du cours ET le tableau 'courses' de l'utilisateur.
— `GET /api/courses/:courseId/students` : Récupère la liste des étudiants inscrits à un cours. Utiliser `.populate('students')`.
— `GET /api/users/:userId/courses` : Récupère la liste des cours auxquels un utilisateur est inscrit. Utiliser `.populate('courses')`.

1.4.5 Gestion des Critiques (Relation 1-to-Many)

— `POST /api/courses/:courseId/reviews` : Ajoute une critique à un cours.
— `GET /api/courses/:courseId/reviews` : Récupère toutes les critiques d'un cours.

1.5 Critères d'Évaluation

— **Modélisation (30%)** : Justesse des schémas et des relations.
— **Fonctionnalité CRUD de base (20%)** : Routes pour User, Profile, Course.
— **Gestion des Relations (40%)** : Routes pour les inscriptions (Many-to-Many) et les critiques (One-to-Many) fonctionnelles, utilisation de `populate`.
— **Qualité du Code et Structure (10%)** : Organisation générale, middlewares, etc.

Pistes de Correction (Extraits de code pour les parties les plus complexes)

1.6 Modélisation des Relations

1.6.1 Relation One-to-One (User et Profile)

```
const profileSchema = new mongoose.Schema({
    user: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'User',
        required: true,
        unique: true
    },
    // ...
});
```

Listing 1 – models/Profile.js

1.6.2 Relation Many-to-Many (User et Course)

C'est une relation à double référence.

```
const userSchema = new mongoose.Schema({
    // ...
    courses: [{
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Course'
    }]
});
```

Listing 2 – models/User.js

```
const courseSchema = new mongoose.Schema({
    // ...
    students: [{
        type: mongoose.Schema.Types.ObjectId,
        ref: 'User'
    }]
});
```

Listing 3 – models/Course.js

1.7 Implémentation des Contrôleurs de Relations

1.7.1 Inscription (Many-to-Many)

```
// @desc    Inscrire un utilisateur à un cours
// @route   POST /api/courses/:courseId/enroll
const enrollUserInCourse = asyncHandler(async (req, res) => {
    const { courseId } = req.params;
    const { userId } = req.body; // On suppose que l'ID de l'utilisateur est dans le body

    const course = await Course.findById(courseId);
    const user = await User.findById(userId);

    if (!course || !user) {
        res.status(404);
        throw new Error('Cours ou utilisateur non trouvé.');
    }

    // Ajout de l'utilisateur au cours (si pas déjà inscrit)
    if (!course.students.includes(userId)) {
        course.students.push(userId);
        await course.save();
    }

    // Ajout du cours à l'utilisateur (si pas déjà ajouté)
    if (!user.courses.includes(courseId)) {
        user.courses.push(courseId);
        await user.save();
    }

    res.status(200).json({ message: 'Inscription réussie.' });
});
```

Listing 4 – controllers/courseController.js - Inscription

1.7.2 Utilisation de `.populate()`

Pour afficher les détails des étudiants au lieu de juste leurs IDs.

```
// @desc    Récupérer les étudiants d'un cours
// @route   GET /api/courses/:courseId/students
const getCourseStudents = asyncHandler(async (req, res) => {
    const course = await Course.findById(req.params.courseId).populate('students', 'username email');
    // .populate('champ', 'champs_a_afficher')

    if (!course) {
        res.status(404);
        throw new Error('Cours non trouvé.');
    }

    res.status(200).json(course.students);
});
```

Listing 5 – controllers/courseController.js - Lister les étudiants

1.7.3 Création d'une Critique (One-to-Many)

```
// @desc    Créer une critique pour un cours
// @route   POST /api/courses/:courseId/reviews
const addReview = asyncHandler(async (req, res) => {
    const { courseId } = req.params;
    const { rating, comment, userId } = req.body;

    const course = await Course.findById(courseId);
    if (!course) {
        res.status(404);
        throw new Error('Cours non trouvé.');
    }

    const review = await Review.create({
        rating,
        comment,
        course: courseId, // Lier la critique au cours
        user: userId
    });

    res.status(201).json(review);
});
```

Listing 6 – controllers/reviewController.js