

Cours MERN - Semaine 3

Persistance des Données : MongoDB, Mongoose et la Programmation Asynchrone

Abdelweheb GUEDDDES & Mohamed Ben Jazia / Ecole Polytechnique Sousse

24 septembre 2025

Table des matières

1	Objectifs Pédagogiques Détaillés	2
2	Partie 1 : Concepts Techniques Approfondis (1h30)	2
2.1	Le Monde Asynchrone de Node.js	2
2.2	MongoDB et Mongoose : Explications Détaillées	3
3	Partie 2 : Intégration Pratique (1h30)	4
3.1	Étape 1 : Création du Cluster sur MongoDB Atlas	4
3.2	Étape 2 : Installation et Configuration dans le Projet	4
3.3	Étape 3 : Création du Modèle Article	5
3.4	Étape 4 : Mise à Jour des Contrôleurs avec <code>async/await</code>	6
3.5	Étape 5 : Mise à Jour du Routeur	7
3.6	Étape 6 : Test Final avec Postman	8
4	Conclusion et Vision pour la Suite	8
5	Travail Pratique Complémentaire (À faire par l'étudiant)	8

1 Objectifs Pédagogiques Détaillés

À la fin de cette séance, vous serez capable de :

- **Expliquer** les principes du NoSQL et les concepts fondamentaux de MongoDB (documents, collections).
- **Maîtriser** le concept de programmation asynchrone en JavaScript, des callbacks aux promesses et à la syntaxe `async/await`.
- **Configurer** un cluster sur MongoDB Atlas et connecter une application Node.js à celui-ci de manière sécurisée.
- **Définir** le rôle d'un ODM comme Mongoose et son utilité.
- **Structurer** des données en créant des Schémas et des Modèles Mongoose avec validation.
- **Implémenter** des opérations de base de données asynchrones dans les contrôleurs en utilisant `async/await` et la gestion d'erreurs `try...catch`.
- **Intégrer** la logique de persistance pour créer et lire des données.

2 Partie 1 : Concepts Techniques Approfondis (1h30)

2.1 Le Monde Asynchrone de Node.js

Toute interaction avec une base de données est une opération d'Entrée/Sortie (I/O). Comme nous l'avons vu, Node.js gère ces opérations de manière **asynchrone** pour ne jamais bloquer le fil d'exécution principal. Gérer cet asynchronisme est au cœur de notre métier de développeur back-end.

Niveau 1 : Les Callbacks La méthode historique. On passe une fonction en argument qui sera exécutée une fois l'opération terminée.

```
1 database.query("SELECT * FROM users", function (err, results)
2 {
3     if (err) { /* Gérer l'erreur */ }
4     // Faire quelque chose avec les 'results'
5 });
```

Listing 1 – Exemple avec un callback (style ancien)

Le problème survient quand on enchaîne plusieurs opérations asynchrones, menant au fameux **"Callback Hell"** (l'enfer des callbacks), un code indenté et difficile à lire.

Niveau 2 : Les Promesses (Promises) Une promesse est un objet qui représente l'achèvement (ou l'échec) éventuel d'une opération asynchrone. Elle peut être dans l'un de ces trois états : *en attente (pending)*, *tenue (fulfilled)*, ou *rompue (rejected)*. On peut enchaîner les opérations avec `.then()` et gérer les erreurs avec `.catch()`.

```
1 database.query("SELECT * FROM users")
2     .then(results => {
3         // Faire quelque chose avec les 'results'
4     })
5     .catch(err => {
6         // Gérer l'erreur
7     });
```

Listing 2 – Exemple avec des promesses

C'est beaucoup plus propre, mais peut devenir lourd avec de longues chaînes.

Niveau 3 : `async/await` (Le Standard Moderne) C'est du "sucre syntaxique" au-dessus des promesses. Cela nous permet d'écrire du code asynchrone qui *ressemble* à du code synchrone, le rendant infiniment plus lisible et facile à maintenir.

Les deux règles d'or de `async/await` :

1. Le mot-clé `await` ne peut être utilisé que **à l'intérieur** d'une fonction déclarée avec le mot-clé `async`.
2. `await` met "en pause" l'exécution de la fonction `async` jusqu'à ce que la promesse soit résolue (ou rejetée), sans pour autant bloquer le programme Node.js.

Toutes les bibliothèques modernes, y compris Mongoose, sont basées sur les promesses, ce qui les rend parfaitement compatibles avec `async/await`. C'est la méthode que nous utiliserons.

2.2 MongoDB et Mongoose : Explications Détaillées

MongoDB est notre lieu de stockage. Il est organisé en **bases de données**, qui contiennent des **collections**. Une collection est un groupe de **documents**, et un document est un objet BSON (similaire à JSON) qui représente une seule entité (un article, un utilisateur, etc.).

Mongoose (l'ODM) est notre traducteur et notre garde-fou. Il se place entre notre application et MongoDB.

- **Le Schéma (Schema)** : C'est le plan de construction de nos documents. On y définit la structure attendue : les noms des champs (`title`, `content`), leur type (`String`, `Number`, `Date`), s'ils sont obligatoires (`required: true`), et des valeurs par défaut (`default: ...`). C'est une couche de validation essentielle pour garantir la cohérence de nos données.
- **Le Modèle (Model)** : Une fois que nous avons un plan (le Schéma), nous pouvons créer un Modèle. Le Modèle est un constructeur, une "usine" à documents, compilée à partir de la définition du Schéma. C'est le Modèle qui nous donne accès à toutes les méthodes pour interagir avec la base de données : `Model.create()`, `Model.find()`, `Model.findById()`, etc.

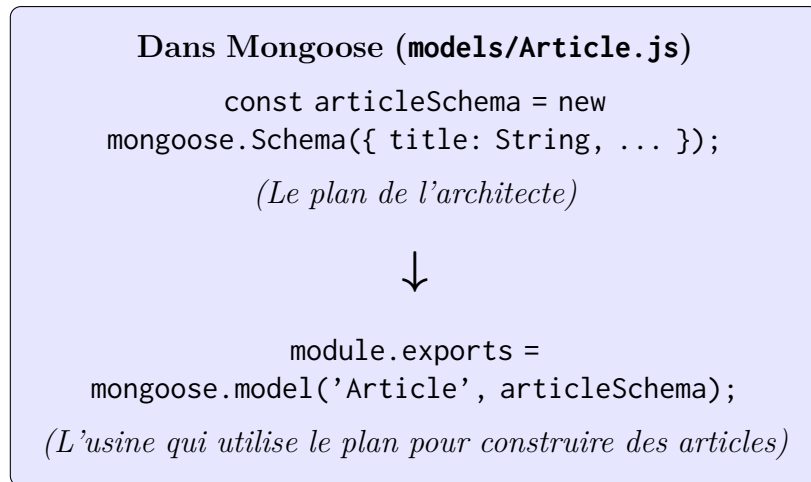


FIGURE 1 – Du Schéma (le plan) au Modèle (l'usine).

3 Partie 2 : Intégration Pratique (1h30)

3.1 Étape 1 : Création du Cluster sur MongoDB Atlas

Cette étape est cruciale et doit être suivie attentivement.

1. Créez un compte sur [MongoDB Atlas](#).
2. Créez un nouveau projet, puis un **cluster gratuit (M0)**.
3. **Configurez la sécurité :**
 - Créez un **utilisateur** de base de données (ex : 'user : strongPassword123').
Notez ces identifiants !
 - Ajoutez une règle d'accès réseau. Pour le développement, choisissez "Allow Access from Anywhere" (0.0.0.0/0).
4. **Récupérez la chaîne de connexion :**
 - Cliquez sur "Connect" → "Connect your application".
 - Copiez la chaîne de connexion. Elle ressemblera à :
`mongodb+srv://<username>:<password>@cluster0.xxxxx.mongodb.net/?retryWrites=true&w=majority`

3.2 Étape 2 : Installation et Configuration dans le Projet

1. **Installez les packages nécessaires** dans votre projet 'mon-api-blog' :

```
1 # Mongoose pour interagir avec MongoDB
2 npm install mongoose
3
4 # Dotenv pour gérer les variables d'environnement
5 npm install dotenv
6
```

2. **Créez un fichier .env** à la racine de votre projet pour stocker votre chaîne de connexion de manière sécurisée. **Ne mettez jamais cette information directement dans le code !**

```
1 # Remplacez les placeholders par VOS informations réelles
2 MONGODB_URI=mongodb+srv://user:strongPassword123@cluster0.xxxxx.
  mongodb.net/blogDB?retryWrites=true&w=majority
```

3

Listing 3 – .env

3. **Créez un fichier de configuration pour la base de données.** C'est une bonne pratique pour centraliser la logique de connexion. Créez config/db.js.

```
1 const mongoose = require('mongoose');
2
3 const connectDB = async () => {
4   try {
5     await mongoose.connect(process.env.MONGODB_URI);
6     console.log('Connexion à MongoDB réussie !');
7   } catch (err) {
8     console.error('Erreur de connexion à MongoDB :', err.
message);
9     // Quitter le processus avec un code d'erreur
10    process.exit(1);
11  }
12 };
13
14 module.exports = connectDB;
15
```

Listing 4 – config/db.js

4. **Appelez la fonction de connexion** dans votre 'server.js'.

```
1 require('dotenv').config(); // Doit être en haut pour charger les
   variables d'env
2 const express = require('express');
3 const connectDB = require('./config/db');
4
5 // Connexion à la base de données
6 connectDB();
7
8 const app = express();
9 const PORT = 3000;
10
11 // ... reste du fichier server.js (middlewares, routes, etc.)
12 // ...
13
```

Listing 5 – server.js - Mise à jour pour la connexion

3.3 Étape 3 : Création du Modèle Article

1. Créez le dossier 'models' s'il n'existe pas.
2. Créez le fichier 'models/Article.js' et définissez le schéma et le modèle.

```
1 const mongoose = require('mongoose');
2
3 const articleSchema = new mongoose.Schema({
4   title: {
5     type: String,
```

```
6         required: [true, 'Le titre est obligatoire.'],
7         trim: true // Enlève les espaces inutiles au début et à
la fin
8     },
9     content: {
10         type: String,
11         required: [true, 'Le contenu est obligatoire.'],
12     },
13     author: {
14         type: String,
15         default: 'Anonyme'
16     },
17     createdAt: {
18         type: Date,
19         default: Date.now
20     }
21 });
22
23 module.exports = mongoose.model('Article', articleSchema);
24
```

Listing 6 – models/Article.js

3.4 Étape 4 : Mise à Jour des Contrôleurs avec `async/await`

C'est ici que nous utilisons la programmation asynchrone pour interagir avec la base de données.

1. Modifiez `controllers/articleController.js`.
2. Importez le modèle 'Article'.
3. Réécrivez la fonction 'createArticle' et ajoutez une nouvelle fonction 'getAllArticles'.

```
1 const Article = require('../models/Article');
2
3 // @desc    Récupérer tous les articles
4 // @route    GET /api/articles
5 const getAllArticles = async (req, res) => {
6     try {
7         // await met en pause la fonction jusqu'à ce que Article.
find() retourne un résultat
8         const articles = await Article.find();
9         res.status(200).json(articles);
10    } catch (err) {
11        // Si une erreur se produit, elle est capturée ici
12        res.status(500).json({ message: "Erreur lors de la récupé
ration des articles.", error: err.message });
13    }
14 };
15
16 // @desc    Créer un nouvel article
17 // @route    POST /api/articles
18 const createArticle = async (req, res) => {
```

```
19 // Le bloc try...catch est essentiel pour gérer les erreurs
    // potentielles
20 // lors des opérations de base de données (ex: validation é
    // chouée).
21 try {
22     const newArticle = new Article({
23         title: req.body.title ,
24         content: req.body.content ,
25         author: req.body.author
26     });
27
28     // await attend que la promesse de .save() soit résolue
29     const savedArticle = await newArticle.save();
30     res.status(201).json(savedArticle);
31
32 } catch (err) {
33     res.status(400).json({ message: "Erreur lors de la cré
        ation de l'article.", error: err.message });
34 }
35 };
36
37 module.exports = {
38     getAllArticles ,
39     createArticle
40 };
41
```

Listing 7 – controllers/articleController.js - avec Mongoose

3.5 Étape 5 : Mise à Jour du Routeur

Assurons-nous que notre routeur utilise bien les nouvelles fonctions du contrôleur.

```
1 const express = require('express');
2 const router = express.Router();
3
4 const { getAllArticles , createArticle } = require('../controllers/
    articleController');
5
6 // Route pour récupérer tous les articles
7 router.get('/', getAllArticles);
8
9 // Route pour créer un nouvel article
10 router.post('/', createArticle);
11
12 module.exports = router;
```

Listing 8 – routes/articleRoutes.js - Mis à jour

3.6 Étape 6 : Test Final avec Postman

1. Lancez votre serveur avec `'npm run dev'`. Observez la console pour le message "Connexion à MongoDB réussie!".
2. **Créez plusieurs articles** avec la requête POST `http://localhost:3000/api/articles`. Variez les données.
3. **Récupérez tous les articles** avec la requête GET `http://localhost:3000/api/articles`. Vous devriez recevoir un tableau JSON contenant tous les articles que vous venez de créer.
4. **Connectez-vous à votre interface MongoDB Atlas** et vérifiez que les documents ont bien été créés dans votre collection `'articles'`.

4 Conclusion et Vision pour la Suite

Félicitations! Vous avez franchi une étape majeure. Votre API n'est plus un jouet ; elle peut maintenant stocker des données de manière persistante, gère l'asynchronisme proprement et possède une structure de code solide.

La semaine prochaine, nous compléterons notre API en implémentant les dernières opérations du CRUD : la lecture d'un seul élément (**Read One**), la mise à jour (**Update**) et la suppression (**Delete**).

5 Travail Pratique Complémentaire (À faire par l'étudiant)

Mettez en pratique ce que vous avez appris en créant le modèle et la logique de base pour les utilisateurs.

1. **Créez un Modèle Utilisateur** (`models/User.js`). Il devra contenir les champs suivants :
 - `'username'` (String, requis, unique)
 - `'email'` (String, requis, unique)
 - `'password'` (String, requis)
2. **Mettez à jour le contrôleur utilisateur** (`controllers/userController.js`) pour utiliser ce nouveau modèle.
 - La fonction `'createUser'` doit maintenant utiliser `User.create()` ou `new User().save()` pour enregistrer un nouvel utilisateur dans la base de données.
 - La fonction `'getAllUsers'` doit utiliser `User.find()` pour récupérer tous les utilisateurs.
3. **Testez de bout en bout** avec Postman : créez plusieurs utilisateurs, puis vérifiez que vous pouvez bien les récupérer.

**Note Importante : Travail à
Rendre** Votre compte rendu doit expliquer en détail
les étapes de création de votre modèle 'User' et la mise à
jour de son contrôleur. Expliquez notamment votre
utilisation de 'async/await' et 'try...catch' dans ce contexte.
Échéance : La soumission doit se faire au plus tard la veille
de la prochaine séance, à 23h59 précises. Ce compte rendu
est obligatoire et noté.