

# Cours MERN - Semaine 5

## Robustesse de l'API : Validation et Gestion des Erreurs

Abdelweheb GUEDDDES & Mohamed Ben Jazia / Ecole Polytechnique Sousse

20 octobre 2025

### Table des matières

<b>1</b>	<b>Objectifs Pédagogiques Détaillés</b>	<b>2</b>
<b>2</b>	<b>Partie 1 : Concepts Techniques Approfondis (1h30)</b>	<b>2</b>
2.1	Finaliser le CRUD : Read One, Update, Delete . . . . .	2
2.2	La Validation : Le Gardien de Vos Données . . . . .	2
2.3	Gestion Centralisée des Erreurs . . . . .	5
2.4	Les Middlewares Express : Le Cœur de l'Application . . . . .	5
2.5	Le Problème du <code>try...catch</code> répétitif . . . . .	5
<b>3</b>	<b>Partie 2 : Atelier Pratique - Robustification de l'API (1h30)</b>	<b>6</b>
3.1	Étape 1 : Amélioration de la Validation du Modèle . . . . .	6
3.2	Étape 2 : Création des Middlewares de Gestion d'Erreurs . . . . .	6
3.3	Étape 3 : Intégration des Middlewares dans <code>server.js</code> . . . . .	7
3.4	Étape 4 : Simplification des Contrôleurs avec <code>express-async-handler</code> . . . . .	8
3.5	Étape 5 : Test final . . . . .	9
<b>4</b>	<b>Conclusion et Vision pour la Suite</b>	<b>9</b>
<b>5</b>	<b>Travail Pratique Complémentaire (À faire par l'étudiant)</b>	<b>9</b>

# 1 Objectifs Pédagogiques Détaillés

Cette semaine, nous transformons notre API fonctionnelle en une API professionnelle et fiable. À la fin de cette séance, vous serez capable de :

- **Expliquer** pourquoi la validation côté serveur est une mesure de sécurité et de cohérence indispensable.
- **Améliorer** les schémas Mongoose avec des validateurs avancés (longueur min/max, expressions régulières, etc.).
- **Comprendre** le concept de middleware dans Express et son fonctionnement (le cycle 'req', 'res', 'next').
- **Créer** des middlewares personnalisés pour gérer des cas spécifiques comme la gestion des routes non trouvées (404).
- **Implémenter** un middleware de gestion d'erreurs global pour centraliser et standardiser les réponses d'erreur de l'API.
- **Éliminer** la répétition des blocs 'try...catch' dans les contrôleurs en utilisant un "wrapper" de fonction asynchrone comme `express-async-handler`.
- **Structurer** le projet pour accueillir les middlewares de manière organisée.

## 2 Partie 1 : Concepts Techniques Approfondis (1h30)

### 2.1 Finaliser le CRUD : Read One, Update, Delete

Nous savons créer et tout lire. Pour une API complète, il faut implémenter :

- **Read One (GET / :id)** : Utilisation de `Model.findById()`. Gestion cruciale du cas où l'ID n'existe pas (retourner une 404).
- **Update (PUT / :id)** : Utilisation de `Model.findByIdAndUpdate()`. Comprendre les options `{ new: true, runValidators: true }`.
- **Delete (DELETE / :id)** : Utilisation de `Model.findByIdAndDelete()`.

### 2.2 La Validation : Le Gardien de Vos Données

Une API robuste ne doit jamais faire confiance aux données envoyées par le client. La validation côté serveur est la première ligne de défense pour assurer :

- **La Sécurité** : Empêche les injections de code malveillant ou de données corrompues.
- **La Cohérence** : Garantit que seules des données valides et conformes à notre modèle métier sont enregistrées dans la base.
- **Une Meilleure Expérience Développeur** : Fournit des messages d'erreur clairs lorsqu'une requête est mal formée.

Mongoose nous offre un système de validation puissant directement dans les schémas. Nous avons déjà vu 'required', mais nous pouvons aller beaucoup plus loin.

```
1 const userSchema = new mongoose.Schema({
2   email: {
3     type: String,
4     required: [true, "L'email est requis"],
5     unique: true, // MongoDB s'assure que chaque email est unique
      via un index
6     lowercase: true, // Convertit l'email en minuscules avant de
      sauvegarder
```

```
7      trim: true,
8      match: [/\\S+@\\S+\\.\\S+/, 'Format d\\'email invalide'] // Valide
le format via Regex
9    },
10   age: {
11     type: Number,
12     min: [18, 'Doit être majeur'],
13     max: [120, 'Age invalide']
14   },
15   role: {
16     type: String,
17     enum: ['user', 'admin'], // Limite les valeurs possibles
18     default: 'user'
19   }
20 });
```

Listing 1 – Exemple de validateurs avancés dans un schéma Mongoose

- **required** : rend le champ obligatoire. Si l'utilisateur ne fournit pas d'adresse e-mail, Mongoose renverra une erreur avant même d'interagir avec la base. Le message d'erreur personnalisé (ici : "L'email est requis") est très utile pour informer le client de manière claire.
- **unique** : assure qu'aucune autre entrée ne possède la même valeur pour ce champ. Attention : il ne s'agit pas d'un validateur logique dans Mongoose, mais de la création d'un **index unique** côté MongoDB. Si deux documents contiennent le même e-mail, la base renverra une erreur de duplication (E11000 duplicate key error). Il est donc conseillé de gérer cette erreur explicitement dans le middleware d'erreurs.
- **lowercase** et **trim** : Ces options permettent de **normaliser les données** avant leur insertion. **lowercase: true** convertit automatiquement la chaîne en minuscules, garantissant une comparaison cohérente (utile pour les e-mails, les noms d'utilisateur, etc.). **trim: true** élimine les espaces superflus, évitant des erreurs subtiles lors des recherches ou comparaisons.
- **match** : applique une expression régulière (regex) pour vérifier la conformité du champ.  
Si la valeur ne correspond pas, Mongoose déclenche une erreur de validation personnalisée avec le message associé.
- **min** et **max** : Ces validateurs numériques contrôlent les bornes d'un champ de type **Number**. Ici, un utilisateur doit être âgé d'au moins 18 ans et d'au plus 120 ans. Ces contraintes garantissent une cohérence métier simple sans ajouter de logique supplémentaire dans le contrôleur.
- **enum** : Restreint les valeurs possibles d'un champ à un ensemble prédéfini. Cela permet d'éviter les incohérences du type "User", "Utilisateur" ou "administrateur" dans la base. Dans ce cas, seul "user" ou "admin" est accepté.
- **default** : Définit une valeur par défaut lorsqu'aucune valeur n'est fournie. Par exemple, si un nouveau compte est créé sans préciser de rôle, il sera automatiquement défini comme "user".

**Les Expressions Régulières (Regex)** Les **expressions régulières** (ou **regex**) sont des motifs utilisés pour rechercher ou valider des chaînes de caractères selon des règles précises. Elles constituent un outil fondamental pour vérifier la conformité d'une donnée

textuelle, comme une adresse e-mail, un mot de passe, ou un numéro de téléphone.

Dans Mongoose (et plus largement en JavaScript), les regex s'écrivent entre deux barres obliques / ... /, et peuvent contenir des symboles spéciaux qui décrivent le motif attendu.

```
1 match: [/^\S+@\S+\.\S+/, "Format d'email invalide"]
```

Listing 2 – Exemple de validation d'un email par regex

Cette ligne signifie :

- `S+` : une ou plusieurs occurrences (+) de tout caractère non espace (`S`).
  - `@` : le symbole arobase obligatoire.
  - `S+` : à nouveau une ou plusieurs occurrences de caractères non espaces (le nom de domaine).
  - `.` : un point littéral (.) — il faut l'échapper avec un antislash car le point seul représente « n'importe quel caractère ».
  - `S+` : enfin, une ou plusieurs lettres après le point, représentant le domaine de haut niveau (`.com`, `.fr`, etc.).
- Ainsi, la regex `/^\S+@\S+\.\S+/` validera des adresses comme :
- `user@example.com`
  - `jean.dupont@mail.fr`
- Mais rejettera des valeurs invalides telles que :
- `userexample.com` (pas d'arobase)
  - `user@com` (pas de point après l'arobase)
  - `user@.com` (nom de domaine vide)

### Principaux symboles utilisés en regex :

Symbole	Signification
<code>.</code>	N'importe quel caractère (sauf le retour à la ligne)
<code>\d</code>	Un chiffre (0–9)
<code>\w</code>	Une lettre, un chiffre ou un underscore ( <code>_</code> )
<code>\s</code>	Un espace ou caractère blanc
<code>\S</code>	Tout sauf un espace
<code>[a-z]</code>	Une lettre minuscule entre a et z
<code>[A-Z]</code>	Une lettre majuscule entre A et Z
<code>[0-9]</code>	Un chiffre entre 0 et 9
<code>+</code>	Une ou plusieurs occurrences
<code>*</code>	Zéro ou plusieurs occurrences
<code>?</code>	Zéro ou une occurrence
<code>^</code>	Début de la chaîne
<code>\$</code>	Fin de la chaîne

## 2.3 Gestion Centralisée des Erreurs

Plutôt que de répéter des blocs `try...catch` identiques dans chaque contrôleur, nous allons introduire un middleware de gestion d'erreurs personnalisé dans Express pour capturer les erreurs (ID invalide, validation Mongoose échouée) et renvoyer des réponses JSON formatées de manière cohérente.

## 2.4 Les Middlewares Express : Le Cœur de l'Application

Un middleware est une fonction qui a accès à l'objet de la requête (`req`), à l'objet de la réponse (`res`), et à la prochaine fonction middleware dans le cycle requête-réponse, communément appelée `next`.

**Le cycle de vie d'une requête avec les middlewares :** Une requête entrante traverse une chaîne de middlewares comme une voiture sur une chaîne de montage. Chaque station (middleware) peut examiner la voiture (la requête), y ajouter ou modifier quelque chose, puis soit l'envoyer à la station suivante (`next()`), soit l'éjecter de la chaîne (envoyer une réponse).

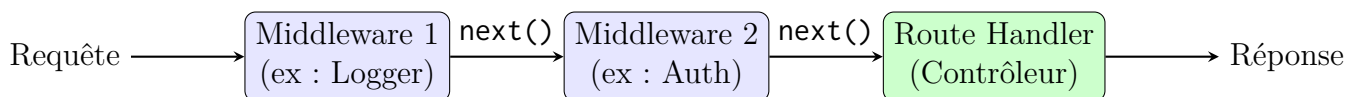


FIGURE 1 – Flux d'une requête à travers une chaîne de middlewares.

Il existe un type spécial de middleware : le **middleware de gestion d'erreurs**. Il est défini avec 4 arguments au lieu de 3 : (`err`, `req`, `res`, `next`). Express l'appellera automatiquement si une erreur est passée à la fonction `next()` (ex : `next(error)`).

## 2.5 Le Problème du `try...catch` répétitif

Dans nos contrôleurs, chaque fonction asynchrone est enveloppée dans un bloc `try...catch`. C'est fonctionnel, mais répétitif et verbeux.

```
1 const myController = async (req, res) => {
2   try {
3     // Logique métier...
4   } catch (err) {
5     res.status(500).json({ message: "Erreur", error: err.message
6   });
7 }
```

Listing 3 – Le schéma répétitif que nous voulons éviter

La solution est d'utiliser un "wrapper" qui s'occupe de ce bloc pour nous. Le package `express-async-handler` est parfait pour cela. Il enveloppe nos fonctions de contrôleur et s'assure que si une erreur se produit, elle est automatiquement passée à notre middleware de gestion d'erreurs global via `next(err)`.

## 3 Partie 2 : Atelier Pratique - Robustification de l'API (1h30)

### 3.1 Étape 1 : Amélioration de la Validation du Modèle

1. Mettez à jour votre modèle **Article.js** pour inclure des validateurs plus stricts.

```
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,
8     minlength: [5, 'Le titre doit contenir au moins 5 caractères.'],
9     maxlength: [100, 'Le titre ne peut pas dépasser 100 caractères.'],
10  },
11  content: {
12    type: String,
13    required: [true, 'Le contenu est obligatoire.'],
14    minlength: [20, 'Le contenu doit contenir au moins 20 caractères.'],
15  },
16  author: {
17    type: String,
18    default: 'Anonyme',
19    trim: true
20  },
21  createdAt: { /* ... */ }
22 });
23
24 module.exports = mongoose.model('Article', articleSchema);
25
```

Listing 4 – models/Article.js - Validation améliorée

2. **Testez la validation** : Utilisez Postman pour essayer de créer un article avec un titre trop court. Vous devriez recevoir une erreur 400 de la part de Mongoose avec un message clair, prouvant que la validation fonctionne.

### 3.2 Étape 2 : Création des Middlewares de Gestion d'Erreurs

1. Créez un nouveau dossier middleware à la racine du projet.
2. Créez un fichier middleware/errorMiddleware.js.

```
1 // Middleware pour les routes non trouvées (404)
2 const notFound = (req, res, next) => {
3   const error = new Error('Route non trouvée - ${req.
4     originalUrl}');
5   res.status(404);
6 }
```

```
5     next(error); // Passe l'erreur au prochain middleware (
    errorHandler)
6 };
7
8 // Middleware de gestion d'erreurs global
9 // Il s'exécute quand une erreur est passée à next()
10 const errorHandler = (err, req, res, next) => {
11     // Parfois, une erreur peut arriver avec un code de succès
    (200), on s'assure que ce ne soit pas le cas.
12     const statusCode = res.statusCode === 200 ? 500 : res.
    statusCode;
13     res.status(statusCode);
14     res.json({
15         message: err.message,
16         // En mode développement, on peut inclure la stack trace
    pour le débogage
17         stack: process.env.NODE_ENV === 'production' ? null : err
    .stack,
18     });
19 };
20
21 module.exports = { notFound, errorHandler };
22
```

Listing 5 – middleware/errorMiddleware.js

### 3.3 Étape 3 : Intégration des Middlewares dans server.js

L'ordre d'intégration est crucial.

1. **Importez les middlewares** dans `server.js`.
2. **Utilisez-les au bon endroit** : le middleware `notFound` doit être placé après toutes vos routes, et le middleware `errorHandler` doit être le tout dernier middleware utilisé.

```
1 // ... (importations express, connectDB, etc.)
2 const { notFound, errorHandler } = require('./middleware/
    errorMiddleware');
3 // ...
4
5 // --- Routes ---
6 app.get('/', (req, res) => { /* ... */ });
7 app.use('/api/articles', articleRoutes);
8 app.use('/api/users', userRoutes); // Si vous l'avez fait
9
10 // --- NOUVEAU : Middlewares de gestion d'erreurs ---
11 // Ce middleware s'exécute si aucune des routes ci-dessus n'a
    correspondu
12 app.use(notFound);
13 // Ce middleware s'exécute si une erreur est passée via next(
    error)
14 app.use(errorHandler);
15
```

```
16 app.listen(PORT, () => {
17   console.log('Serveur démarré sur http://localhost:${PORT}');
18 });
19
```

Listing 6 – server.js - Intégration des middlewares

### 3.4 Étape 4 : Simplification des Contrôleurs avec **express-async-handler**

#### 1. Installez le package :

```
1 npm install express-async-handler
2
```

#### 2. Refactorisez votre contrôleur d'articles (controllers/articleController.js).

- Importez `asyncHandler`.
- "Enveloppez" chaque fonction de contrôleur avec `asyncHandler`.
- Supprimez tous les blocs `try...catch`.

```
1 const asyncHandler = require('express-async-handler');
2 const Article = require('../models/Article');
3
4 // @desc    Récupérer tous les articles
5 // @route   GET /api/articles
6 const getAllArticles = asyncHandler(async (req, res) => {
7   const articles = await Article.find();
8   res.status(200).json(articles);
9 });
10
11 // @desc    Créer un nouvel article
12 // @route   POST /api/articles
13 const createArticle = asyncHandler(async (req, res) => {
14   const { title, content, author } = req.body;
15
16   // Une petite validation manuelle en plus de Mongoose
17   if (!title || !content) {
18     res.status(400);
19     throw new Error('Veuillez fournir un titre et un contenu.
20   ');
21   }
22
23   const article = await Article.create({ title, content, author });
24   res.status(201).json(article);
25 });
26 // ... refactorisez aussi getArticleById, updateArticle,
27 // deleteArticle de la même manière
28 module.exports = {
29   getAllArticles,
30   createArticle,
31   // ...

```

```
32 };  
33
```

Listing 7 – controllers/articleController.js - Refactorisé

### 3.5 Étape 5 : Test final

1. **Testez la route 404** : Essayez d'accéder à une URL qui n'existe pas, comme `/api/nonexistent`. Vous devriez obtenir une réponse JSON standardisée de votre middleware `notFound`.
2. **Testez la validation** : Essayez de créer un article avec un titre trop court. Vous devriez obtenir une réponse JSON standardisée de votre `errorHandler`.
3. **Testez une erreur d'ID** : Essayez de récupérer un article avec un ID mal formé (ex : `/api/articles/123`). `Mongoose` va lever une erreur de "cast", qui sera capturée par `asynchandler` et formatée par votre `errorHandler`.

## 4 Conclusion et Vision pour la Suite

Votre API est maintenant non seulement fonctionnelle, mais aussi robuste. Elle valide les entrées, gère les erreurs de manière cohérente et son code est plus propre et plus lisible. Vous avez franchi une étape majeure dans la construction d'applications de qualité professionnelle.

Ceci conclut la première grande partie de notre cours : la construction d'une API back-end solide. À partir de la semaine prochaine, nous basculons côté client et nous allons commencer à construire l'interface qui consommera cette API avec **React**.

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

Appliquez les concepts de cette semaine à la ressource "Utilisateurs".

1. **Améliorez la validation** dans votre modèle `User.js` en utilisant les validateurs vus en cours (ex : `unique`, `match` pour l'email, `minlength` pour le mot de passe).
2. **Refactorisez** entièrement votre `UserController.js` en utilisant `express-asynchandler` pour supprimer tous les blocs `try...catch`.
3. **Ajoutez une validation manuelle** dans le contrôleur `createUser` pour vérifier que l'email et le mot de passe sont bien présents avant de tenter de créer l'utilisateur. Si ce n'est pas le cas, levez une erreur avec `throw new Error('...')`.
4. **Testez rigoureusement** avec Postman : essayez de créer un utilisateur avec un email invalide, un mot de passe trop court, ou un email déjà existant pour voir comment votre `errorHandler` répond.

## Note Importante : Travail à Rendre

Votre compte rendu doit détailler la refactorisation de votre contrôleur utilisateur. Expliquez comment **express-async-handler** interagit avec votre middleware **errorHandler**. Incluez des captures d'écran des tests de validation sur Postman. 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.