Rapport - Projet MongoDB

Décembre 2024

Paradigmes BD

Étudiants: EVEN Tom, JAURAS Maxime,

LEROUX Mathieu, POSSENTI Alexandre

Table des matières

Introduction : **2**
Technologies Utilisées : **2**
1. MongoDB : 2
2. React : 2
3. Tailwind CSS : 2
4. Node.js et Express.js : 2
5. Axios : 3
Fonctionnalités clés : **4**
Création de la BD : **4**
Données 4
Validateur de données 5
Index **6**
Backend **7**
1. Configuration du serveur : 7
2. Connexion à MongoDB : 7
3. Initialisation : 7
Gestion de la récupération d'un produit par référence dans l'API 8
Exemple de requêtes "delete" 9
Exemple de requêtes "update et create" 9
Frontend **10**
Visuel du site 10

Introduction :

Nous avons décidé de ne pas nous baser sur l'exemple donné en cours, mais de concevoir notre propre application de gestion de stocks de marchandises. Cette application permet d'obtenir la liste complète des articles avec leurs informations (nom, prix, quantité disponible, etc.). Les fonctionnalités incluent également l'ajout, la suppression et la modification des articles.

Technologies Utilisées :

1. MongoDB :

Pour la gestion de notre base de données, nous avons choisi **MongoDB**, une base de données NoSQL orientée documents. MongoDB nous permet de stocker nos données sous forme de documents JSON, ce qui facilite leur manipulation et leur mise à jour.
Afin d'améliorer les performances et l'accessibilité, nous avons utilisé **MongoDB Atlas**, une plateforme cloud qui nous permet d'héberger notre cluster MongoDB, rendant la base de données accessible depuis n'importe où.

2. React :

Pour la conception de l'interface utilisateur et la gestion des interactions front-end, nous avons opté pour le framework **React**. Ce framework JavaScript moderne offre une architecture basée sur les composants, facilitant le développement, la réutilisation de code et la maintenance de notre application.

3. Tailwind CSS :

Pour styliser l'interface utilisateur, nous avons utilisé **Tailwind CSS**, un framework CSS. Cet outil nous a permis de créer une interface claire et intuitive.

4. Node.js et Express.js :

Pour le backend, nous avons utilisé **Node.js** associé à **Express.js**, un framework minimaliste pour Node.js. Cette combinaison nous a permis de construire des API RESTful efficaces et maintenables, facilitant les interactions entre le frontend et la base de données.
Express.js simplifie la gestion des routes, des middlewares et des requêtes HTTP, ce qui nous a permis de gagner du temps dans le développement et d'assurer une architecture backend robuste.

5. Axios :

Pour la gestion des requêtes HTTP dans notre application, nous avons choisi Axios, une bibliothèque JavaScript promise-based. Axios simplifie les appels API grâce à une syntaxe claire et concise, ce qui permet d'envoyer et de recevoir des données de manière efficace entre le frontend et le backend.

L'utilisation d'Axios nous a permis de :

- Faciliter la gestion des requêtes grâce à ses méthodes intuitives (GET, POST, PUT, DELETE), rendant le code plus lisible.
- Gérer les réponses et les erreurs avec un système intégré de promesses, ce qui améliore le contrôle des flux asynchrones dans notre application.
- Configurer globalement les en-têtes pour les requêtes HTTP, ce qui est particulièrement utile pour intégrer des tokens d'authentification ou des paramètres personnalisés.
- Optimiser les requêtes grâce à l'interception : nous avons utilisé les interceptors d'Axios pour centraliser le traitement des requêtes et des réponses, comme l'ajout automatique d'un token ou la gestion des erreurs globales.

Cette approche nous a permis d'assurer une communication fluide et sécurisée entre le frontend développé en React et notre backend basé sur Node.js et Express.js.

Fonctionnalités clés :

- Visualisation de la liste des articles en stock avec leurs informations.
- Ajout de nouveaux articles via une interface utilisateur simple.
- Modification des informations des articles existants (prix, nom, quantité, etc.).
- Suppression des articles devenus obsolètes ou inutiles.

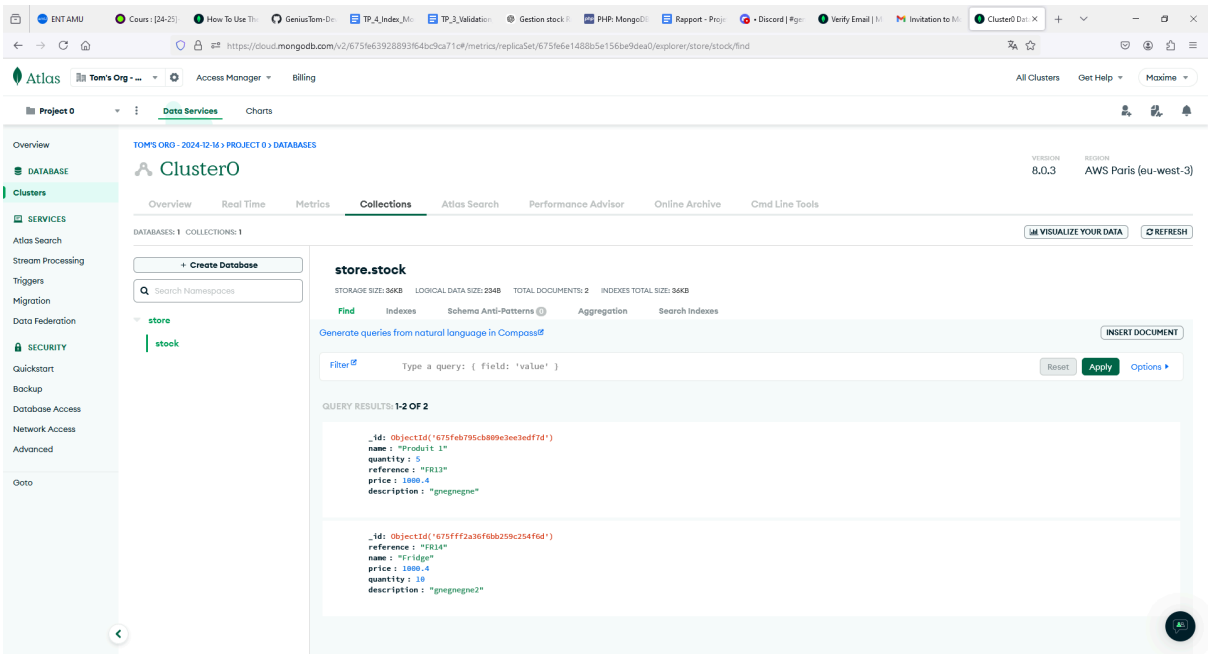
En associant ces technologies, nous avons réussi à créer une application performante, adaptée aux besoins de gestion de stocks.

Création de la BD :

Données

La base de données contient une seule collection, appelée **“stock”**, qui regroupe les informations relatives aux articles enregistrés. Cette collection inclut plusieurs champs saisis par le gérant lors de l’enregistrement des articles :

- **name** : le nom de l’article.
- **quantity** : le nombre d’unités disponibles pour cet article.
- **price** : le prix de l’article.
- **description** : des informations supplémentaires concernant l’article.
- **reference** : un identifiant unique pour chaque produit.



Validateur de données

Ce code définit un validateur JSON pour la collection **“stock”**. Il impose les champs obligatoires suivants :

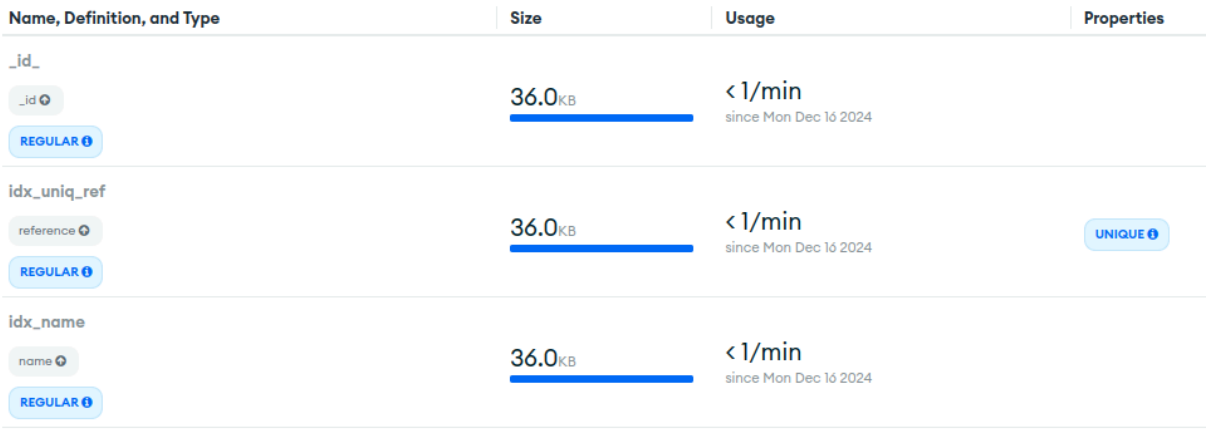
- **reference** : une chaîne respectant le motif [a-z][a-z][0-9]{2}.
- **name** : une chaîne (nom du produit).
- **price** : un nombre (double) ≥ 0.
- **quantity** : un entier (int) ≥ 0.

Les autres propriétés, comme **tags** (tableau de chaînes), sont facultatives.

```
{
    "collMod": "stock",
    "validator": {
        $jsonSchema: {
            "bsonType": "object",
            "required": ["reference", "name", "price", "quantity", ],
            "properties": {
                "reference" : {
                    "bsonType" : "string",
                    "pattern": "[A-Z][A-Z][0-9]{2}",
                    "description": "Unique Reference of the product - required."
                },
                "name": {
                    "bsonType": "string",
                    "description": "Name of the product - required."
                },
                "price": {
                    "bsonType": "double",
                    "minimum": 0,
                    "description": "Price of the product - required"
                },
                "quantity": {
                    "bsonType": "",
                    "minimum": 0,
                }
            }
        }
    }
}
```

Index

Afin d'optimiser nos recherches et de renforcer les contraintes de validité de nos documents. Nous avons en premier lieu créé un index unique sur le champ référence afin de garantir son unicité, ce qui nous permet d'utiliser ce champ pour identifier un produit de manière unique afin d'effectuer des modifications et suppressions.
Nous avons également créé un index 'idx_name' qui nous permettra d'optimiser les recherches effectuées sur le nom des produits.



On peut donc voir ici les deux index que nous avons créés ainsi que l'index sur le champ '_id' qui est généré automatiquement par MongoDB.

Backend

Cette fonction initialise un serveur Express pour interagir avec une base de données MongoDB.

```
const express = require('express');
const { MongoClient, Double, ObjectId} = require('mongodb');
const cors = require('cors'); // Importer cors
const app = express();
const PORT = 5013;

app.use(cors());
app.use(express.json());


const uri = "mongodb+srv://tomeven:I1FolNn3GV5r6ZOx@cluster0.dd0qz.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0";
const client = new MongoClient(uri);

let collection = null

async function getCollection() {
    await client.connect();
    const database = client.db('store');
    return database.collection('stock');
}
```

1. **Configuration du serveur :**
    - Création de l'application Express.
    - Activation du middleware **CORS** pour permettre les requêtes provenant d'autres origines.
    - Activation du middleware pour analyser les requêtes en format JSON.

2. **Connexion à MongoDB :**
    - Utilisation de l'URI pour se connecter à un cluster MongoDB distant via MongoClient.
    - Une fonction getCollection est définie pour établir une connexion à la base de données **"store"** et accéder à la collection **"stock"**.

3. **Initialisation :**

Une variable collection est prête à recevoir la collection MongoDB pour éviter de reconnecter la base de données à chaque requête.

Gestion de la récupération d’un produit par référence dans l’API

Cette fonction configure une route **GET** qui permet de récupérer un produit en fonction de sa référence. Si la référence est manquante, une erreur 400 est renvoyée. La fonction recherche ensuite le produit dans la base de données et, si trouvé, renvoie ses informations sous format JSON. Si le produit n’est pas trouvé, une erreur 404 est renvoyée, et en cas d’erreur interne, une erreur 500 est générée. Enfin, le serveur démarre et écoute les requêtes sur un port spécifique.

```
app.get('/products', async (req, res) => {
    try {
        if (collection === null) {
            collection = await getCollection();
        }

        const find = req.query.filter ? JSON.parse(req.query.filter) : {};
        const sort = req.query.sort ? JSON.parse(req.query.sort) : {};
        const page = parseInt(req.query.page) || 1;
        const limit = parseInt(req.query.limit) || 50;

        const products = await collection.find(find).sort(sort).skip((page-1)*limit).limit(limit).toArray()


        if (products.length === 0) {
            return res.status(404).send('Produit non trouvé');
        }

        res.json(products); // Renvoie le produit trouvé
    } catch (error) {
        console.error(error);
        res.status(500).send('Erreur lors de la récupération du produit.');
    }
});
```

Exemple de requêtes “delete”

Cette route **DELETE** supprime un produit en utilisant un filtre fourni dans la requête ; si le filtre est absent, elle retourne une erreur 400, si aucun produit ne correspond, une erreur 404, et en cas d’erreur interne, un statut 500.

```
app.delete('/deleteOne', async (req, res) => {
    try {
        if (collection === null) {
            collection = await getCollection();
        }

        const filter = req.body.filter ? req.body.filter : "";
        if(req.body.filter === "") {
            return res.status(400).send('Paramètre filter manquant');
        }
        const products = collection.deleteOne(filter);

        if (products.length === 0) {
            return res.status(404).send('Produit non trouvé');
        }
        res.json(products); // Renvoie le produit trouvé
    } catch (error) {
        console.error(error);
        res.status(500).send('Erreur lors de la récupération du produit.');
    }
});
```

Exemple de requêtes “create et update”

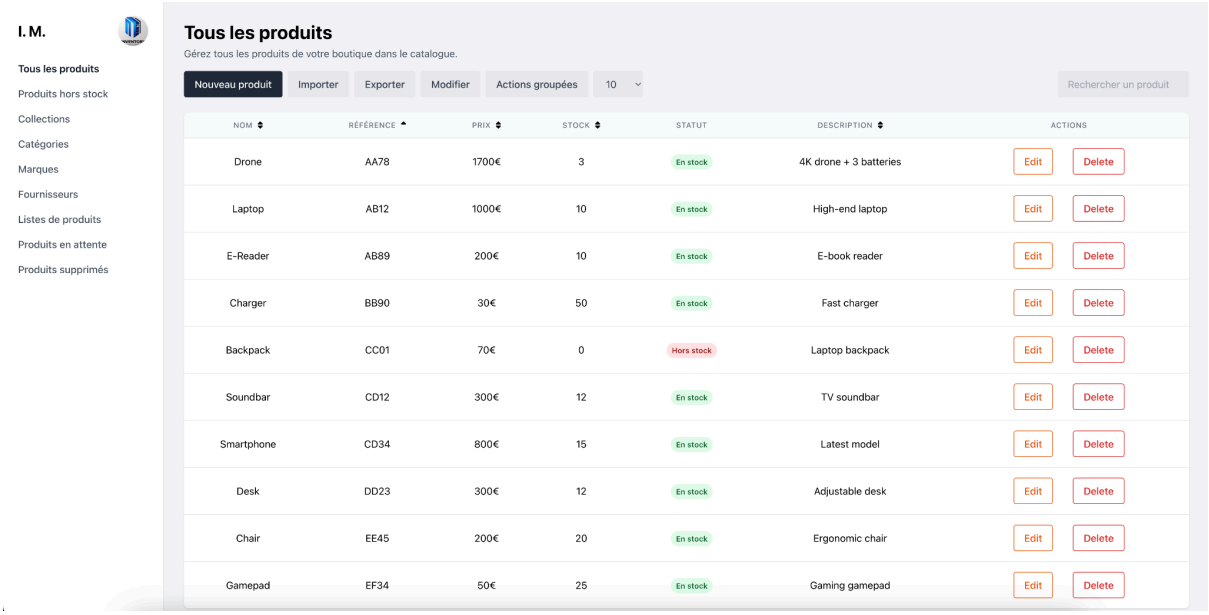
```
app.post('/addProduct', async (req, res) => {
    try {
        if (collection === null) {
            collection = await getCollection();
        }
        const product = req.body.product;
        product.price = new Double(product.price);


        const result = await collection.insertOne(product);
        res.json(result);
    } catch (error) {
        let errorDetails = [];
        for (let i = 0; i < error.errorResponse.errInfo.details["schemaRulesNotSatisfied"][0]["propertiesNotSatisfied"].length; i++) {
            const errorObject = error.errorResponse.errInfo.details["schemaRulesNotSatisfied"][0]["propertiesNotSatisfied"][i]
            errorDetails.push(errorObject.propertyName
                + " : " + errorObject.details[0].reason
                + ". Attempted value : "
                + errorObject.details[0].specifiedAs.bsonType
                + ". Considered type : "
                + errorObject.details[0].consideredType)
        }
        console.log(errorDetails)
        res.status(500).send(errorDetails);
    }
});
```

```
app.put('/updateOneProduct', async (req, res) => {
    try {
        if (collection === null) {
            collection = await getCollection();
        }
        let filter = req.body.filter;

        let updateSet = req.body.updateSet;
        delete updateSet._id;

        updateSet.price = new Double(updateSet.price);

        console.log(filter)
        console.log(updateSet)
        const result = await collection.updateOne(filter, { $set: updateSet });
        console.log(result)
        res.json(result);
    } catch (error) {
        let errorDetails = [];
        for (let i = 0; i < error.errorResponse.errInfo.details["schemaRulesNotSatisfied"][0]["propertiesNotSatisfied"].length; i++) {
            const errorObject = error.errorResponse.errInfo.details["schemaRulesNotSatisfied"][0]["propertiesNotSatisfied"][i]
            errorDetails.push(errorObject.propertyName
                + " : " + errorObject.details[0].reason
                + ". Attempted value : "
                + errorObject.details[0].specifiedAs.bsonType
                + ". Considered type : "
                + errorObject.details[0].consideredType)
        }
        console.log(errorDetails)
        res.status(500).send('Error updating one product.');
    }
});


        for (let i = 0; i < error.errorResponse.errInfo.details["schemaRulesNotSatisfied"][0]["propertiesNotSatisfied"].length; i++) {

        }
        console.log(errorDetails)
        res.status(500).send(errorDetails);
    }
});
```

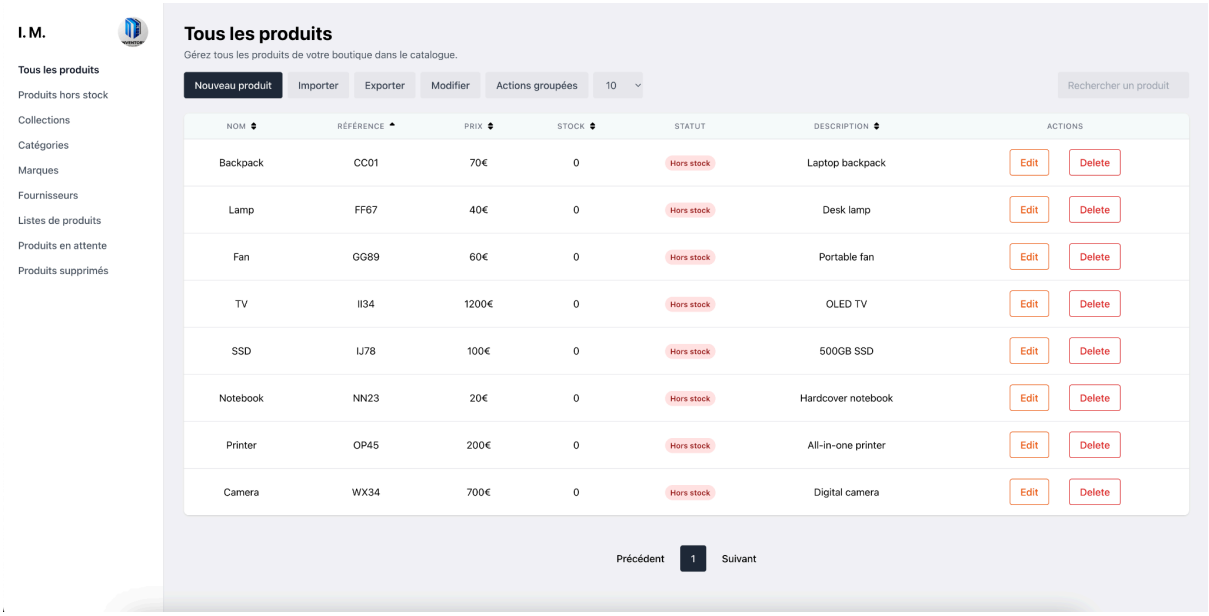
Frontend

Visuel du site

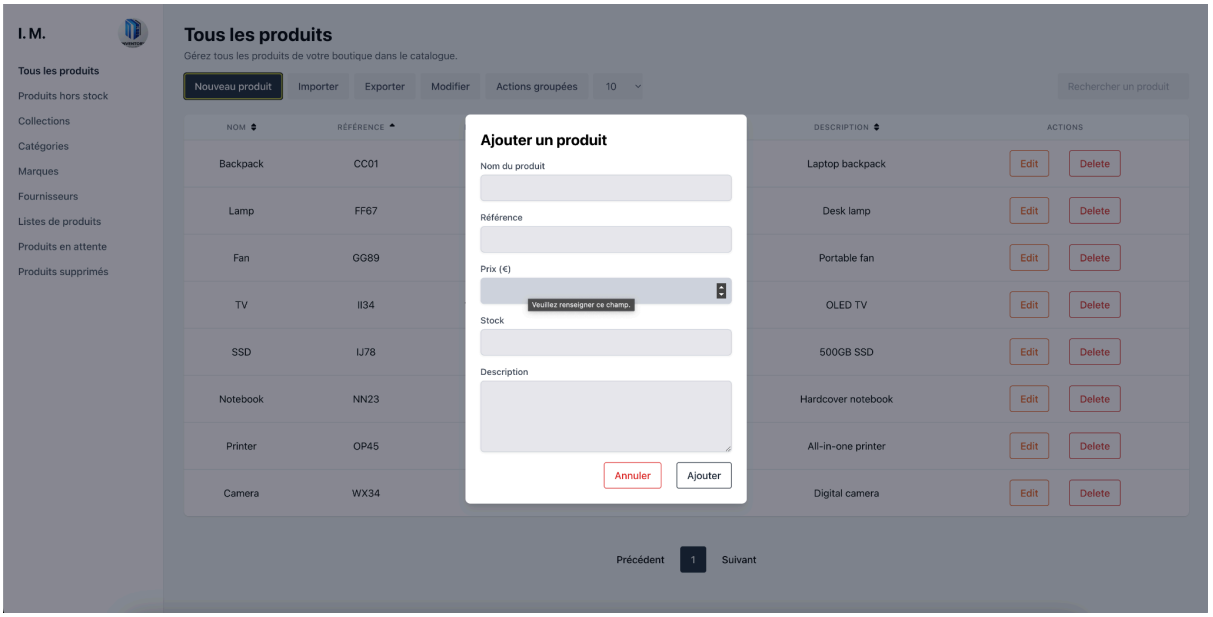
Ici, nous avons la liste de tous les articles :



On observe ici qu’on a filtré et affiché tous les articles en rupture de stock :



Page pour ajouter un produit :



Page pour modifier un produit préalablement sélectionné :