Codelab: Connexion à MongoDB et persistance avec NodeJS

MongoDB est une base de données NoSQL populaire qui se concentre sur les hautes performances et la disponibilité. MongoDB implémente un modèle de document. Le modèle de document autorise les données qui n'ont pas de structure rigide. Les documents comprennent des paires clé-valeur.

Bien que MongoDB soit généralement identifié comme une base de données NoSQL, il fournit une syntaxe de type SQL. Dans ce codelab, nous allons implémenter une API en utilisant MongoDB.

Base de données MongoDB dockerisée

Cette partie concerne uniquement ceux qui sont intéressés par l'utilisation d'une version dockerisée de MongoDB. Si vous avez déjà installé MongoDB localement ou en utilisant MongoDB Atlas, vous pouvez ignorer cette partie.

Vous pouvez démarrer un serveur MongoDB exécutant la dernière version de MongoDB à l'aide de Docker avec la commande suivante :

```
$ docker run --publish 27017:27017 --name node-mongo --detach mongo:latest
```

Cela extraira la dernière image officielle de Docker Hub. L'ajout de l'indicateur –detach garantira que le conteneur Docker s'exécute en tant que processus d'arrière-plan, distinct du shell. La balise -p indique le port auquel le port de conteneur est lié à 27017. Vous pouvez vous connecter à MongoDB sur localhost:27017.

Pour plus d'informations :

- https://earthly.dev/blog/mongodb-docker/
- https://www.geeksforgeeks.org/how-to-run-mongodb-as-a-docker-container/

Préparation de la collection “books”

Assurez-vous que la collection de livres (**books.json**) que nous avons utilisée au cours des semaines précédentes se trouve dans votre base de données MongoDB et est accessible (https://cours-info.iut-bm.univ-fcomte.fr/upload/supports/S4/BDD/R4.04%20Au%20del%20du%20relationnel%20JAZAR%20et%20CDEHE/TD_sem5.zip). Vous pouvez ignorer cette partie si la base de données est prête.

La collection json "**books.json**" peut être ajoutée manuellement à une base de données en utilisant MongoDB Compass ou en utilisant la shell (mongoimport). Dans MongoDB Compass, il suffit de sélectionner la base de données cible, de cliquer sur le bouton "Ajouter une collection" et de choisir le fichier json à importer. Dans la shell, il faut d'abord se connecter à la base de données cible, puis exécuter la commande mongoimport en spécifiant le nom de la collection et le chemin du fichier json à importer. Cette opération peut être effectuée en utilisant la ligne de commande du système d'exploitation.

Pour les utilisateurs de Windows :

- https://www.mongodb.com/community/forums/t/import-data-into-mongodb-using-cmd-for-windows-os-users-on-local-database/145939

Il est important de noter que pour pouvoir ajouter manuellement la collection "books.json" à une base de données en utilisant MongoDB Compass ou la shell, il faut d'abord **s'assurer que le service MongoDB est démarré et en cours d'exécution**. Si le service n'est pas en cours d'exécution, il faut le démarrer avant d'effectuer l'opération d'importation.

Création d'un projet NodeJS

Créez un nouveau projet **mongo_intro** et initialisez Node:

```
$ mkdir mongo_intro
$ cd mongo_intro
$ npm init --yes
```

Pour se connecter à une base de données MongoDB à l'aide de NodeJS et de la bibliothèque mongodb, il est nécessaire d'installer tout d'abord la bibliothèque mongodb en utilisant la commande npm suivante :

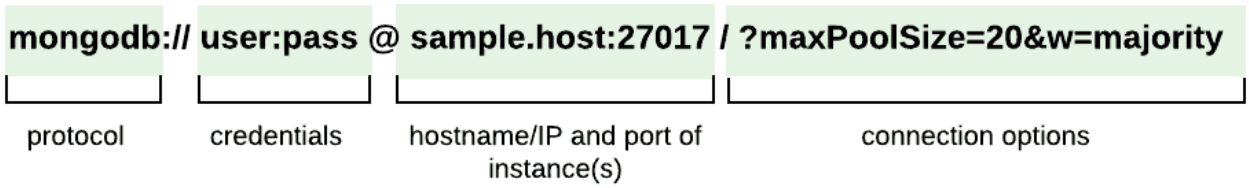
```
$ npm install mongodb
```

Créez un fichier nommé **tasks.js** ; cela contiendra notre code d'application qui interagit avec une collection MongoDB nommée "tasks":

```
$ touch tasks.js
```

Connexion au serveur

L'URI de connexion est l'ensemble d'instructions que le pilote mongo (driver) utilise pour se connecter à un déploiement MongoDB. Il indique au pilote comment il doit se connecter à MongoDB et comment il doit se comporter lorsqu'il est connecté. L'exemple suivant montre chaque partie de l'URI de connexion :

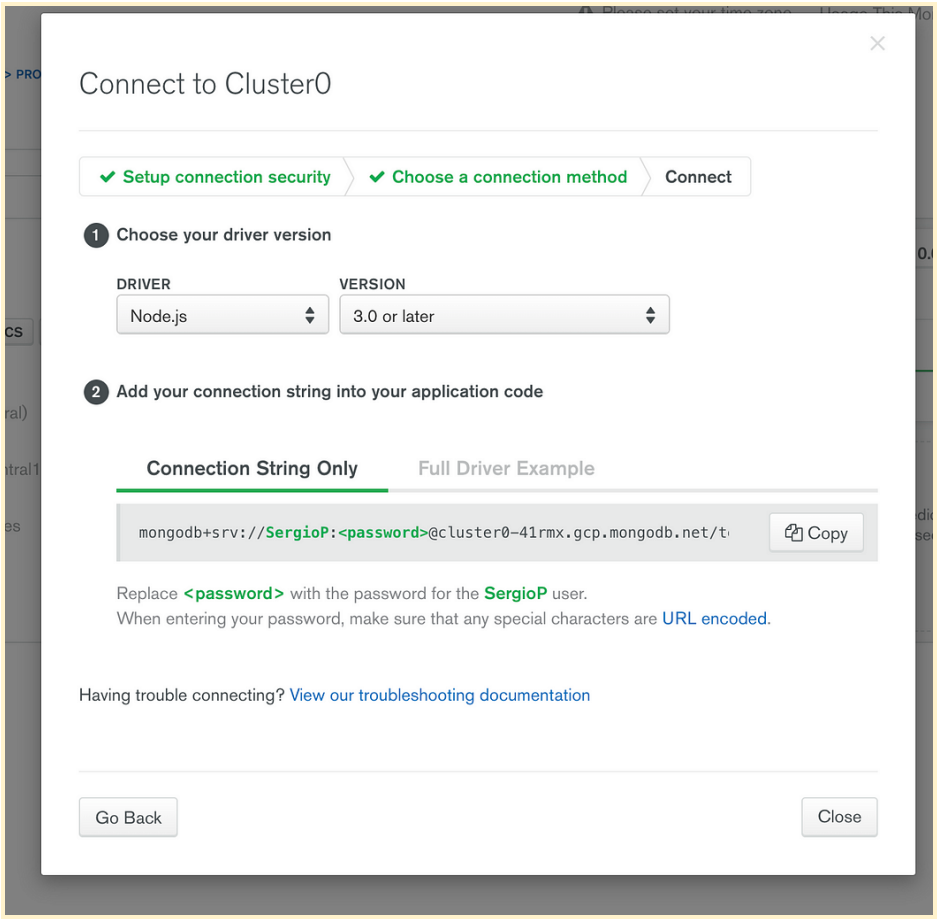


- https://www.mongodb.com/docs/manual/reference/connection-string/#std-label-connections-standard-connection-string-format

Pour notre MongoDB local autonome (standalone), nous pourrions utiliser cet URI :

- mongodb://localhost:27017

Si vous utilisez MongoDB Atlas, revenez à la page d'accueil de MongoDB Atlas et sélectionnez l'option Clusters. Cliquez sur l'option "Connect Your Application" pour configurer une connexion entre votre application et le cluster à l'aide des pilotes natifs de MongoDB. Sélectionnez Node.js comme pilote natif et enregistrez la chaîne de connexion fournie par MongoDB Atlas pour référence ultérieure.



Le code ci-dessous montre comment vous pouvez utiliser un URI de connexion dans un client pour vous connecter à MongoDB. Copiez ce code dans le fichier **tasks.js** (assurez-vous de modifier l'URI si vous avez une chaîne de connexion différente ou si vous utilisez MongoDB Atlas).

```
const { MongoClient } = require("mongodb");
const URI = "mongodb://localhost:27017";

// Créer un nouveau MongoClient
const client = new MongoClient(URI);

async function run() {
   try {
       // Connecter le client au serveur
       await client.connect();
       // Établir et vérifier la connexion
       await client.db("db_r403").command({ ping: 1 });
       console.log("Connecté avec succès au serveur");
   } finally {
       // Garantit que le client se fermera lorsque vous aurez terminé/en cas
d'erreur
       await client.close();
   }
}
run().catch(console.dir);
```

Exécutez le code :

```
$ node tasks.js
```

Ce code renverra en sortie le message "**Connecté avec succès au serveur**" si la connexion à la base de données a réussi. La ligne **await client.db("db_r403").command({ ping : 1 });** va tester la connexion à la base de données. Assurez-vous de changer le nom de la base de données si vous avez un autre nom que "**db_r403**".

```
(base) josephazar@Josephs-MBP-2 mongo_intro % node tasks.js
Connecté avec succès au serveur
```

Nous allons maintenant créer deux fonctions : **addTask()** et **listTasks()**. Nous allons prendre un nom de tâche du terminal, l'enregistrer dans la collection "**tasks**", puis afficher toutes les tâches de la collection dans le terminal.

La fonction **listTasks()** se connectera à la base de données, obtiendra toutes les tâches et les imprimera dans la console.

```
async function listTasks() {
   try {
       await client.connect();
       const myDB = client.db("db_r403"); // BDD ⟹ db_r403
       const myColl = myDB.collection("tasks"); // collection ⟹ tasks
       const result = await myColl.find(); // ⟹ db.tasks.find()
       await result.forEach(console.dir);
   } finally {
       await client.close();
   }
}
```

La fonction **addTask(task)** prendra une tâche comme argument et insérera un document avec cette tâche puis imprimera le **_id** dans la console. Après cela, elle appellera la fonction **listTasks()**.

```
async function addTask(task) {
   try {
       await client.connect();
       const myDB = client.db("db_r403");
       const myColl = myDB.collection("tasks");

       // L'exemple suivant utilise la méthode insertOne() pour
       // insérer un nouveau document dans la collection de tâches:
       const result = await myColl.insertOne({task:task});
       console.log(
           `Un document a été inséré avec le _id: ${result.insertedId}`,
       );
   } finally {
       await client.close();
       await listTasks();
   }
}
```

Le processus de l'ensemble du programme est le suivant : l'utilisateur exécute le script NodeJS avec un argument de ligne de commande (la tâche). Le programme vérifie si nous pouvons nous connecter à la base de données. Ensuite, le programme insérera la tâche et affichera les documents si la tâche est fournie (**addTask**), sinon il affichera les documents de la collection “tasks” (**listTasks**).

```
const { MongoClient } = require("mongodb");
const URI = "mongodb://localhost:27017";
// Créer un nouveau MongoClient
const client = new MongoClient(URI);
async function run() {
    try {
        // Connecter le client au serveur
        await client.connect();
        // Établir et vérifier la connexion
        await client.db("db_r403").command({ ping: 1 });
        console.log("Connecté avec succès au serveur");
    } finally {
        // Garantit que le client se fermera lorsque vous aurez terminé/en cas d'erreur
        await client.close();
    }
}
async function addTask(task) {
    try {
        await client.connect();
        const myDB = client.db("db_r403");
        const myColl = myDB.collection("tasks");

        // L'exemple suivant utilise la méthode insertOne() pour
        // insérer un nouveau document dans la collection de tâches:
        const result = await myColl.insertOne({task:task});
        console.log(
            `Un document a été inséré avec le _id: ${result.insertedId}`,
        );
    } finally {
        await client.close();
        await listTasks();
    }
}
async function listTasks() {
    try {
        await client.connect();
        const myDB = client.db("db_r403"); // BDD ==> db_r403
        const myColl = myDB.collection("tasks"); // collection ==> tasks
        const result = await myColl.find(); // ==> db.tasks.find()
        await result.forEach(console.dir);
    } finally {
        await client.close();
    }
}
async function main(task){
    await run().catch(console.dir); // Testez la connexion
    if (task) {
        await addTask(task);
    } else {
        await listTasks();
    }
}
const task = process.argv[2]; // Prendre l'argument de la ligne de commande
main(task);
```

```
(base) josephazar@Josephs-MBP-2 mongo_intro % node tasks.js "Pratiquer MongoDB pour passer le module R403"
Connecté avec succès au serveur
Un document a été inséré avec le _id: 640e1893c1b09ebd09f1cb5b
{
  _id: ObjectId {
    [Symbol(id)]: Buffer(12) [Uint8Array] [
      100,  14,  24, 147, 193,
      176, 158, 189,   9, 241,
      203,  91
    ]
  },
  task: 'Pratiquer MongoDB pour passer le module R403'
}
```

```
test> use db_r403
switched to db db_r403
db_r403> show collections
books
tasks
db_r403> db.tasks.find()
[
  {
    _id: ObjectId("640e1893c1b09ebd09f1cb5b"),
    task: 'Pratiquer MongoDB pour passer le module R403'
  }
]
```

```
(base) josephazar@Josephs-MBP-2 mongo_intro % node tasks.js "Pratiquer NodeJS pour passer le module R401"
Connecté avec succès au serveur
Un document a été inséré avec le _id: 640e18fd3c6619173c39dcf6
{
  _id: ObjectId {
    [Symbol(id)]: Buffer(12) [Uint8Array] [
      100,  14,  24, 147, 193,
      176, 158, 189,   9, 241,
      203,  91
    ]
  },
  task: 'Pratiquer MongoDB pour passer le module R403'
}
{
  _id: ObjectId {
    [Symbol(id)]: Buffer(12) [Uint8Array] [
      100,  14, 24, 253, 60,
      102,  25, 23,  60, 57,
      220, 246
    ]
  },
  task: 'Pratiquer NodeJS pour passer le module R401'
}
```

Créer une API avec Express et Mongoose

Mongoose est une bibliothèque de modélisation de données d'objets qui vous permet d'appliquer des schémas à vos données MongoDB avec Node.js. Cela vous évite d'avoir à valider manuellement vos objets de document. Essayons de modéliser la liste des livres à l'aide de Mongoose.
Installez le module **mongoose** à partir du registre npm :

```
$ npm install mongoose
```

Créez un fichier nommé **books.js**

```
$ touch books.js
```

Créez un répertoire **models** et un fichier dedans **book.js** :

```
$ mkdir models
```

```
$ touch models/book.js
```

Ce fichier (**book.js**) va vous montrer comment modéliser la collection de livres avec Mongoose. Tout dans Mongoose commence par un schéma. Chaque schéma correspond à une collection MongoDB et définit la forme des documents au sein de cette collection.

Pour définir le schéma des livres, nous devons d'abord comprendre la structure de chaque document JSON dans le fichier **books.json**.

Copiez ce qui suit dans le fichier **models/book.js** :

```
const mongoose = require('mongoose');
const { Schema } = mongoose;
mongoose.connect('mongodb://localhost:27017/db_r403');
const db = mongoose.connection;
db.on('error', (err) => {
   console.error('Erreur de connexion MongoDB:', err);
});
db.once('open', () => {
   console.log('Connexion MongoDB réussie');
});
const bookSchema = new mongoose.Schema({
   _id: Number,
   title: String,
   isbn: String,
   pageCount: Number,
   publishedDate: Date,
   thumbnailUrl: String,
   shortDescription: String,
   longDescription: String,
   status: String,
   authors: [String],
   categories: [String],
});
const Book = mongoose.model('book', bookSchema);
module.exports = Book;
```

Ce modèle Mongoose représente la collection "**books**" dans la base de données MongoDB. Par défaut, Mongoose utilisera le nom du modèle au singulier et créera automatiquement une collection avec la forme plurielle du nom du modèle. Dans ce cas, comme le nom du modèle est "**book**", Mongoose recherchera une collection nommée "**books**" dans la base de données connectée.

Ajoutez maintenant ce qui suit au fichier **books.js** :

```
const Book = require('./models/book');
async function main(){
   let books = await Book.find({}, '_id title isbn authors categories')
       .sort({ _id: 1 })
       .limit(3);
   console.log(books)
}
main();
```

Cela récupérera tous les documents de la collection de livres, les triera par **_id** dans l'ordre croissant, limitera les résultats aux 3 premiers et les enregistrera dans la console. Notez que nous utilisons **sort({ _id: 1 })** pour trier les résultats par **_id** dans l'ordre croissant, où 1 représente l'ordre croissant et -1 représente l'ordre décroissant.

Testez le code :

```
$ node books.js
```

```
(base) josephazar@Josephs-MBP-2 mongo_intro % node books.js
Connexion MongoDB réussie
[
  {
    _id: 1,
    title: 'Unlocking Android',
    isbn: '1933988673',
    authors: [ 'W. Frank Ableson', 'Charlie Collins', 'Robi Sen' ],
    categories: [ 'Open Source', 'Mobile' ]
  },
  {
    _id: 2,
    title: 'Android in Action, Second Edition',
    isbn: '1935182722',
    authors: [ 'W. Frank Ableson', 'Robi Sen' ],
    categories: [ 'Java' ]
  },
  {
    _id: 3,
    title: 'Specification by Example',
    isbn: '1617290084',
    authors: [ 'Gojko Adzic' ],
    categories: [ 'Software Engineering' ]
  }
]
```

Notez que cette requête ne sélectionne que les champs "**_id**", "**title**", "**isbn**", "**authors**" et "**categories**". Ceci est défini dans la deuxième partie de la méthode **find()**. Si vous souhaitez obtenir tous les champs, vous pouvez supprimer cette option de la requête.
Dans le dossier racine, créons un nouveau fichier **server.js** :

```
$ npm install express cors
```

```
$ touch server.js
```

```
$ mkdir routes
```

```
$ mkdir controllers
```

```
$ touch routes/books.router.js
```

```
$ touch controllers/books.controller.js
```

Dans ce qui suit, nous utiliserons le pattern Routeur-Contrôleur-Modèle. Implémentez le fichier **server.js** comme suit :

```
// server.js
const express = require("express");
const cors = require("cors");
const booksRoutes = require("./routes/books.router");
const app = express();
var corsOptions = {
   origin: "http://localhost:3000"
};
app.use(cors(corsOptions));
// parser les requêtes de type de contenu - application/json
app.use(express.json());
// parser les requêtes de type de contenu - application/x-www-form-urlencoded
app.use(express.urlencoded({ extended: true }));
// route simple
app.get("/", (req, res) => {
   res.json({ message: "MongoDB - NodeJS TD R403" });
});
// route /api/books
app.use("/api/books",booksRoutes);
const PORT = 3000;
app.listen(PORT, () => {
   console.log(`Le serveur écoute sur le port ${PORT}.`);
});
```

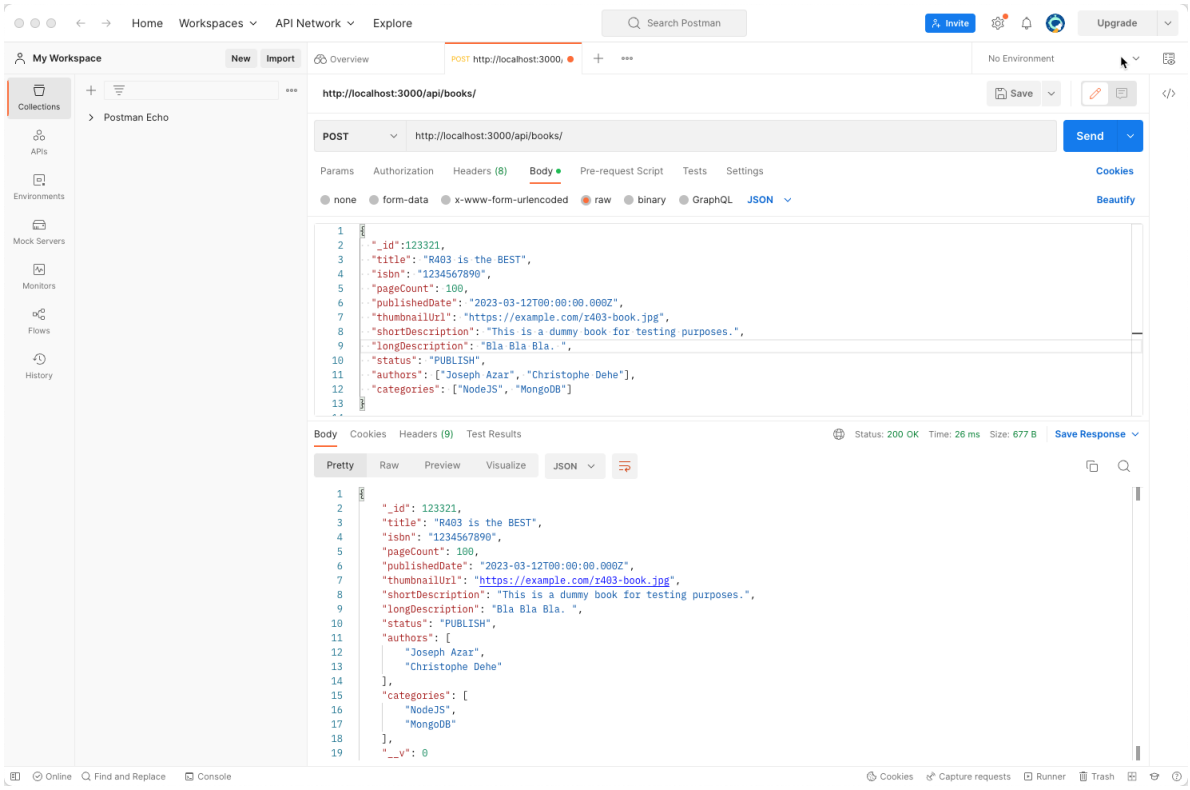
Dans le fichier du routeur, implémentez les routes suivantes :

Route	Méthode	Opération
api/books/	POST	créer un nouveau livre
api/books/	GET	récupérer tous les livres
api/books/published	GET	récupérer tous les livres publiés (status = PUBLISH)
api/books/:id	PUT/PATCH	mettre à jour la description courte d'un livre avec un identifiant donné
api/books/:id	DELETE	supprimer un livre avec un identifiant donné

Dans le fichier du contrôleur, implémentez la logique de chaque opération. Vous trouverez ci-dessous la requête **Mongoose** pour chaque opération :

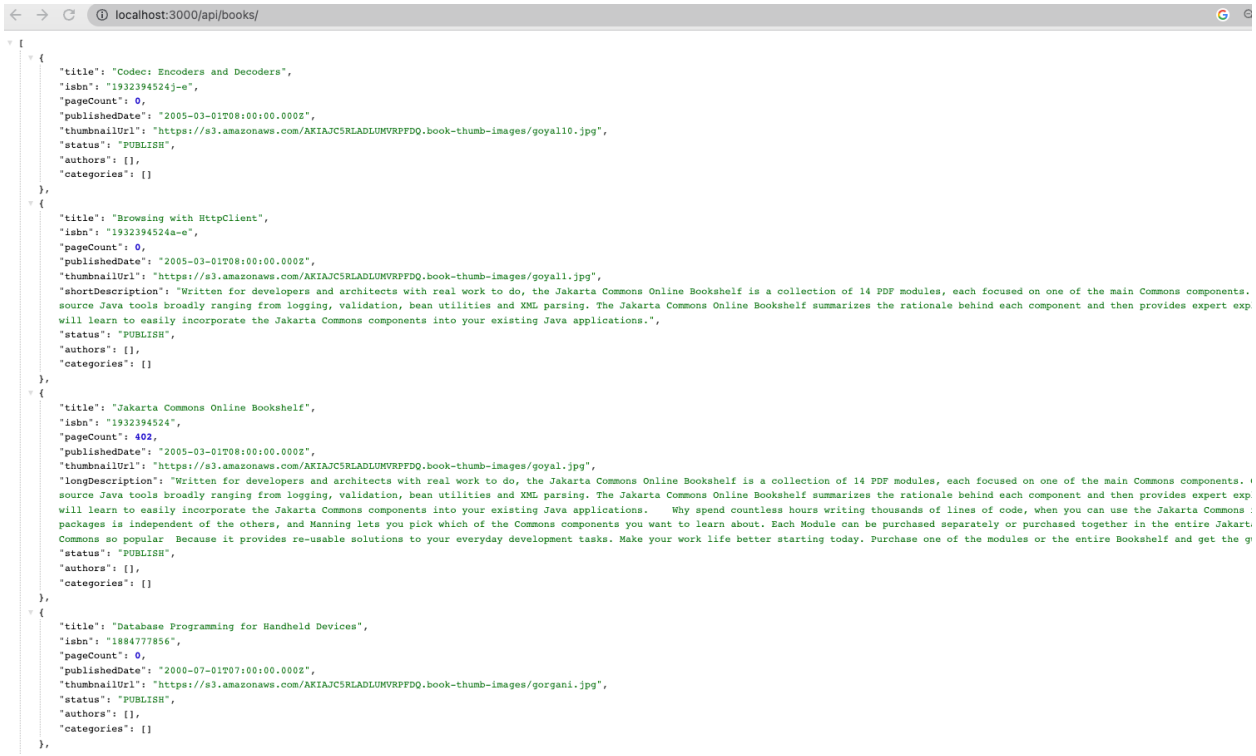
Créer un nouveau livre

```
const newBook = new Book(bookData); // bookData ⟹ JSON DOCUMENT
// Enregistrez le nouveau document de livre dans la base de données
newBook.save(newBook).then(data ⇒ {
   res.status(200).send(data);
}).catch(err ⇒ {
       res.status(500).send({
           message:
               err.message || "Error creating book."
       });
   });
```



Récupérer tous les livres

```
Book.find()
  .sort({ _id: -1 })
  .then((books) => {
      res.status(200).send(books);
  }).catch(err => {
  res.status(500).send({message: err.message || "Error retrieving books."
  });
});
```

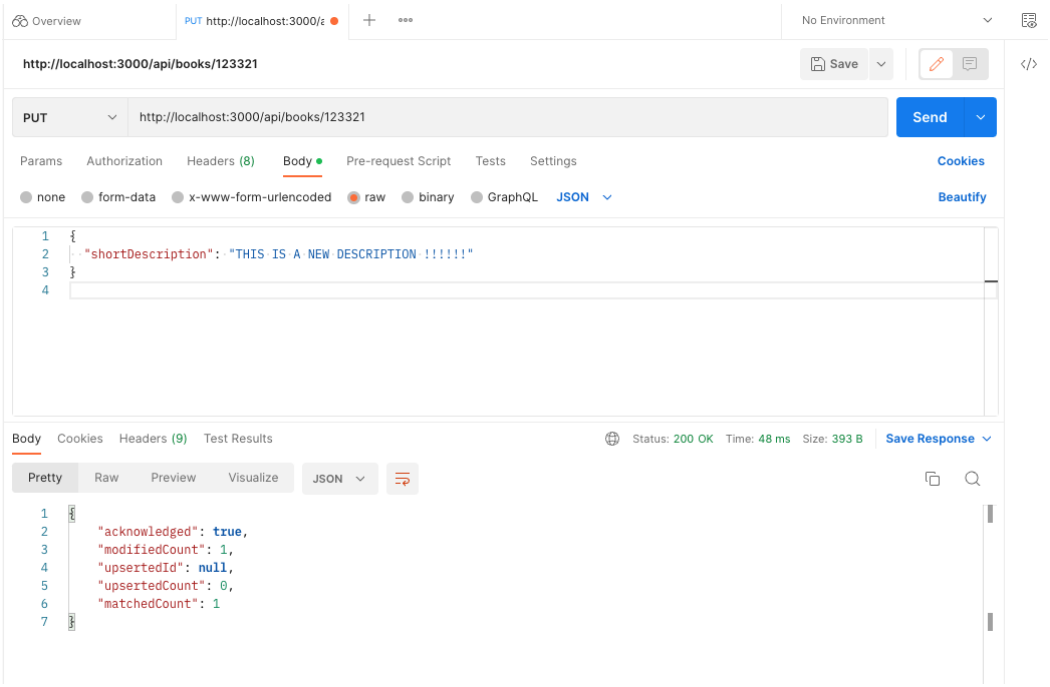


Récupérer tous les livres publiés (status = PUBLISH)

```
Book.find({ status: 'PUBLISH' })
   .sort({ _id: -1 })
   .then((books) => {
       res.status(200).send(books);
   }).catch(err => {
   res.status(500).send({message: err.message || "Error retrieving books."});
});
```

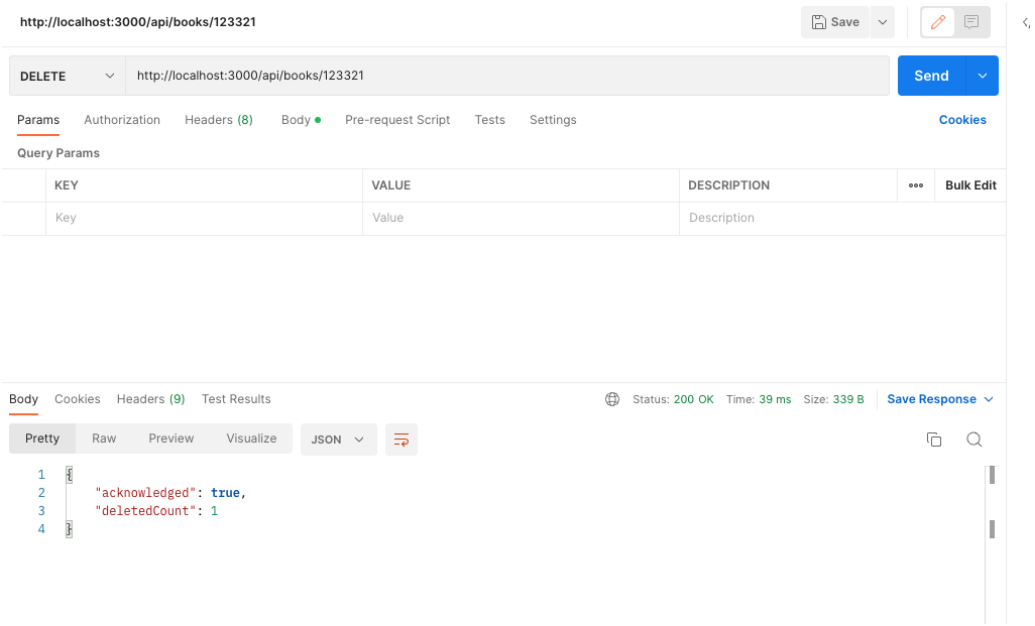
Mettre à jour la description courte d'un livre avec un identifiant donné

```
Book.updateOne({ _id: id }, { shortDescription: shortDescription
}).then((result) => {
   res.status(200).send(result);
}).catch(err => {
   res.status(500).send({message: err.message || "Error updating book."});
});
```



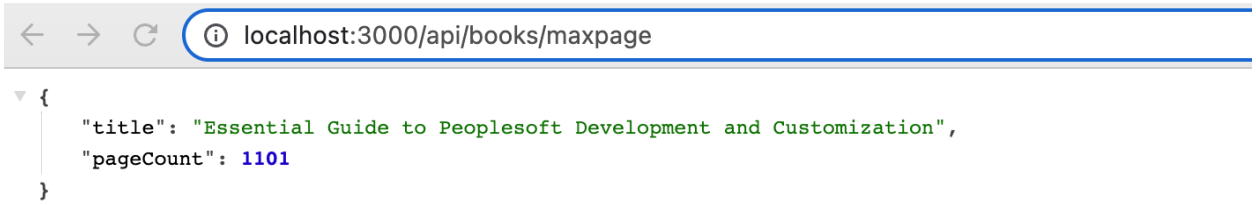
Supprimer un livre avec un identifiant donné

```
Book.deleteOne({ _id: id }).then((result) => {
   res.status(200).send(result);
}).catch(err => {
   res.status(500).send({message: err.message || "Error deleting book."});
});
```



Agrégation avec Mongoose

Supposons que nous voulions obtenir le livre qui a le nombre maximum de pages. Pour cela, on peut faire une agrégation sur le champ pageCount de la collection “books”.



Vous pouvez suivre le raisonnement suivant dans MongoShell pour obtenir la sortie souhaitée. Vous pouvez sûrement aussi le faire en une seule requête.

```
db_r403> var maxPage = db.books.aggregate([ { $group: { _id: null, maxPageCount: { $max: '$pageCount' } } }, { $project: { _id: 0, maxPageCount: 1 } }]).toArray()[0].maxPageCount

db_r403> maxPage
1101
db_r403> db.books.find({"pageCount": maxPage},{_id:0,title:1})
[
  {
    title: 'Essential Guide to Peoplesoft Development and Customization'
  }
]
```

Afin d'implémenter un contrôleur qui fait l'équivalent, vous pouvez utiliser ce code :

```
exports.findMaxPage = async (req,res) => {
    try{
        let result = await Book.aggregate([
            { $group: { _id: null, maxPageCount: { $max: '$pageCount' }}},
            { $project: { _id: 0, maxPageCount: 1 }}
        ]);

        let title = await Book.findOne({"pageCount":result[0].maxPageCount},
                                       {_id:0,title:1})
        res.status(200).send({title: title.title, pageCount:
result[0].maxPageCount});
    }catch (e){
        res.status(500).send({message: e.message || "Internal Error."});
    }
}
```

Implémentez la route et le contrôleur correspondant pour tester le code ci-dessus.

Pour aller plus loin, vous pouvez essayer d'implémenter les exercices d'agrégation du TD (semaine 5 : TD intro à MongoDB) en utilisant Mongoose.