

Introduction

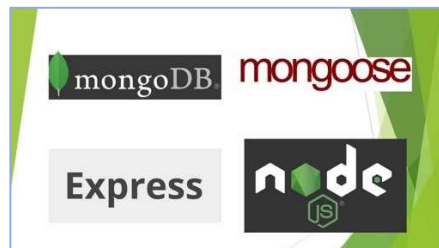
Node.JS est une technologie très développée ces dernières années et le langage JavaScript a évolué avec son temps. Désormais, il semble incontournable de savoir créer une application utilisant Node.JS. Nous allons, dans cet atelier, vous montrer comment créer simplement un serveur **ExpressJS** en **JavaScript**.



Express.js est un Framework minimaliste pour node.js. Il permet de créer facilement une application web. Son côté minimaliste le rend peu pratique pour créer des applications de taille importante. Nous allons nous en servir pour développer des applications jouant un rôle de serveur de données pour nos applications en back end.

Avec Node.js, on n'utilise pas de serveur web HTTP comme Apache. En fait, c'est à nous de créer le serveur.

D'autre part, **MongoDB** est un système de gestion de base de données orienté documents, de la classe des bases de données NoSQL.

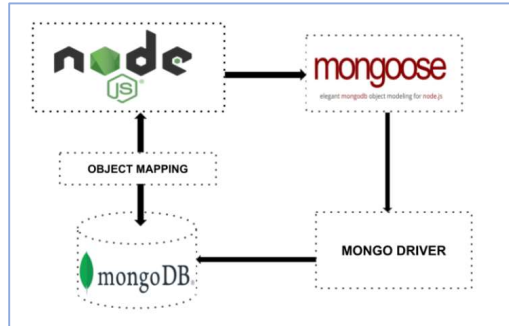


Appelée également « Not Only SQL » (pas seulement SQL), la base de données NoSQL est une approche de la conception des bases et de leur administration particulièrement utile pour de très grands ensembles de données distribuées.

Dans MongoDB, les données sont modélisées sous forme de documents sous un style JSON. On ne parle plus de tables, ni d'enregistrements mais de collections et de documents.

Tout document appartient à une collection et a un champ appelé **_id** qui identifie le document dans la base de données.

Mongoose est une bibliothèque de programmation JavaScript orientée objet qui crée une connexion entre MongoDB et le Framework d'application Web Express.



Mongoose est un pilote Node.JS pour MongoDB. Il offre certains avantages par rapport au pilote MongoDB par défaut, comme l'ajout de types aux schémas. Une différence est que certaines requêtes Mongoose peuvent différer de leurs équivalents MongoDB.

Prérequis :

Installer NodeJs (obligatoire)

Accédez au site <https://nodejs.org/en/download/>

Installer MongoDB

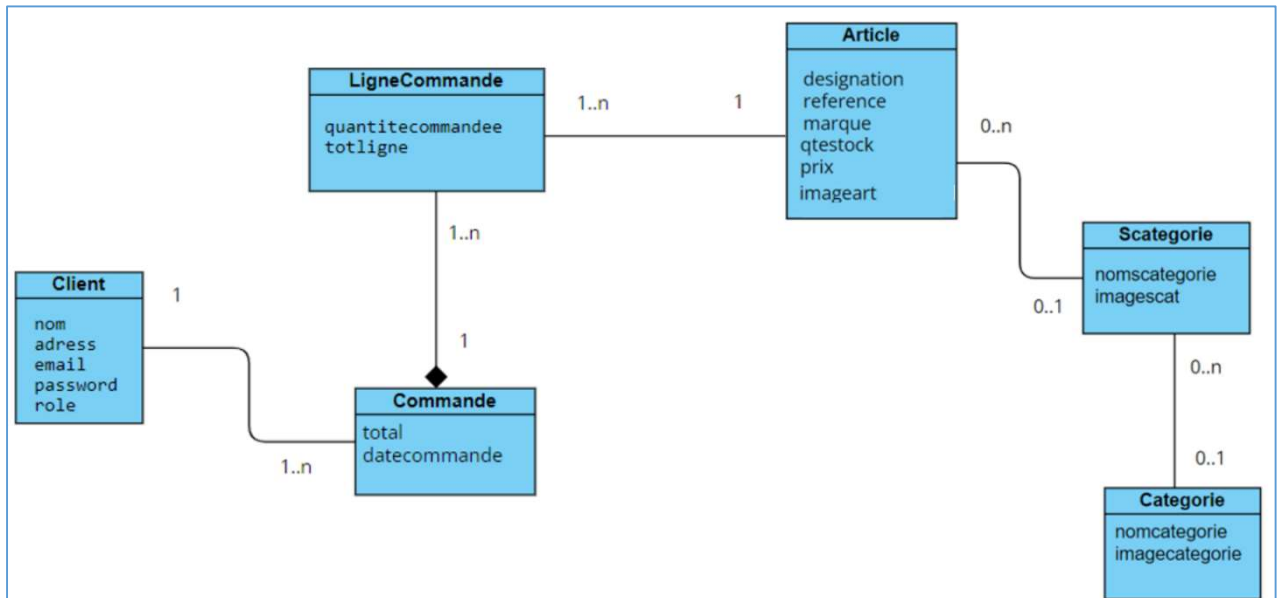
Accédez au site <https://www.mongodb.com/docs/manual/installation/>

Etude ce cas : Site de Commerce en ligne

Notre objectif est de voir comment construire et créer une API CRUD REST en utilisant node.js express et MongoDB avec mongoose. Une API est une interface logicielle qui permet à deux applications de communiquer entre elles. En d'autres termes, une API est un messenger qui envoie votre demande au fournisseur, puis vous renvoie la réponse.



Lors de la mise en place des modèles, on considèrera la base de données déduite d'une partie du diagramme de classes UML :



1. Tout d'abord, on doit créer un nouveau projet pour gérer le back end de notre application.

```

mkdir ecommerce\backend
cd ecommerce\backend
c:\ecommerce\backend>npm init -y

```

Un package.json vient d'être généré contenant des valeurs par défaut.

2. Démarrer l'application avec visual studio code

```
code .
```

3. Faire l'installation des dépendances suivantes :

```
npm i express mongoose dotenv cors
```

Toute application a besoin d'installer des packages. Le premier est évidemment express qui est un Framework web léger pour Node.js que nous avons utilisé pour aider à construire notre serveur back-end. Nous allons installer dans cet atelier cors, dotenv et mongoose.

cors signifie partage de ressources cross-origin et nous permet d'accéder à des ressources en dehors de notre serveur à partir de notre serveur.

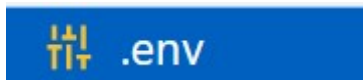
dotenv nous facilitera l'utilisation d'un fichier .env pour stocker des variables sensibles telles que le port ou le nom de la base de données.

mongoose nous aidera à simplifier l'interaction avec MongoDB dans Node.js.

```
npm i -g nodemon
```

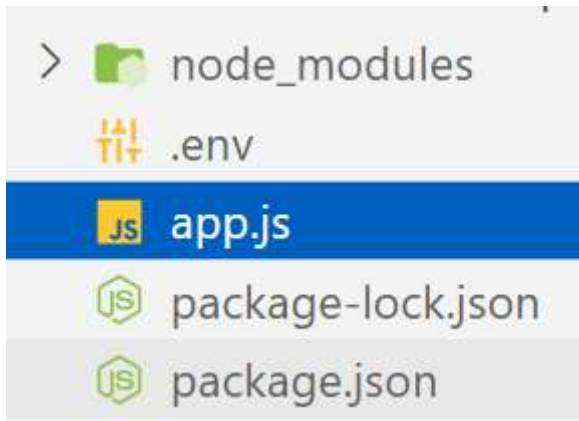
nodemon est un utilitaire d'interface de ligne de commande (CLI). Il enveloppe votre application Node, surveille le système de fichiers et redémarre automatiquement le processus.

- Commencer par préparer les variables d'environnement dans un fichier intitulé **.env**



```
DATABASE=mongodb://127.0.0.1:27017/DatabaseCommerce
PORT=3001
```

- Mettre le code suivant dans un nouveau fichier qu'on va appeler **app.js** :



```
const express=require('express');
const mongoose =require("mongoose")
const dotenv =require('dotenv')
const cors = require('cors')

const app = express();

//config dotenv
dotenv.config()

//Les cors
app.use(cors())

//BodyParser Middleware
app.use(express.json());

// Connexion à la base données
mongoose.connect(process.env.DATABASE)
    .then(() => {console.log("DataBase Successfully Connected");})
    .catch(err => { console.log("Unable to connect to database", err)};
process.exit(); });

// requête
app.get("/",(req,res)=>{
res.send("bonjour");
```

```
});  
  
app.listen(process.env.PORT, () => {  
  console.log(`Server is listening on port ${process.env.PORT}`); });  
module.exports = app;
```

Un module est une bibliothèque/fichier JavaScript que vous pouvez importer dans un autre code en utilisant la fonction **require()** de Node. Express lui-même est un module, tout comme les bibliothèques de middleware et de base de données que nous utilisons dans nos applications Express.

Le code ci-dessus montre comment nous importons un module par son nom, en utilisant le Framework Express comme exemple. Tout d'abord, nous invoquons la fonction **require()**, en spécifiant le nom du module sous forme de chaîne ('express'), et en appelant l'objet retourné pour créer une application Express. Nous pouvons alors accéder aux propriétés et fonctions de l'objet application.

La fonction **express.json()** est une fonction middleware intégrée dans Express. Il analyse les requêtes entrantes avec des charges utiles JSON et est basé sur l'analyseur de corps.

Une application Express est fondamentalement une série de fonctions appelées middleware. Chaque élément de middleware reçoit les objets request et response, peut les lire, les analyser et les manipuler, le cas échéant. Le middleware Express reçoit également la méthode next, qui permet à chaque middleware de passer l'exécution au middleware suivant.

Nous nous connectons à MongoDB avec la méthode **mongoose.connect()**.

La fonction **app.listen()** est utilisée pour lier et écouter les connexions sur l'hôte et le port spécifiés.

L'application démarre un serveur et écoute le port 3001 à la recherche de connexions. L'application répond « bonjour » aux demandes adressées à l'URL racine (/) ou à la route racine.

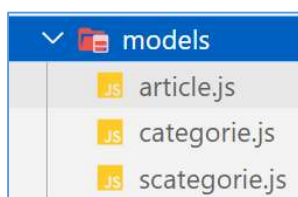
Exécuter le code avec la commande

```
nodemon app
```

```
[nodemon] 2.0.15  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching path(s): *.*  
[nodemon] watching extensions: js,mjs,json  
[nodemon] starting `node app.js`  
Server is listening on port 3001  
DataBase Successfully Connected
```

6. Créer les modèles

Créer le dossier models dans lequel on mettra tout notre modèle de base de données



models/categorie.js

```
const mongoose =require("mongoose")
const categorieSchema=mongoose.Schema({
  nomcategorie:{ type: String, required: true,unique:true },
  imagecategorie :{ type: String, required: false }
})
module.exports=mongoose.model('categorie',categorieSchema)
```

models/scategorie.js

```
const mongoose =require("mongoose")
const Categorie =require("./categorie.js");
const scategorieSchema=mongoose.Schema({
  nomscategorie:{ type: String, required: true },
  imagescat :{ type: String, required: false },
  categorieID: {type:mongoose.Schema.Types.ObjectId,
  ref:Categorie}
})

module.exports=mongoose.model('scategorie',scategorieSchema)
```

Dans notre Model, il contient un champ qui est défini sur le type **ObjectId**, et avec l'option **ref**, nous avons dit à mongoose d'utiliser l'identifiant de notre Model pour faire reference à Catégorie.

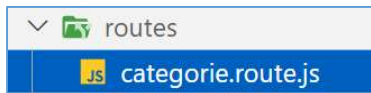
models/article.js

```
const mongoose =require("mongoose")
const Scategorie =require("./scategorie.js");
const articleSchema=mongoose.Schema({
  reference:{ type: String, required: true,unique:true },
  designation:{ type: String, required: true,unique:true },
  prix:{ type: Number, required: false },
  marque:{ type: String, required: true },
  qtestock:{ type: Number, required: false },
  imageart:{ type: String, required: false },
  scategorieID: {type:mongoose.Schema.Types.ObjectId,
```

```
ref:Scategorie}  
  
})  
  
module.exports=mongoose.model('article',articleSchema)
```

I. CRUD categories

1. On va maintenant créer les routes pour la classe **categorie** sous le dossier à créer routes.



routes/categorie.route.js

```
var express = require('express');  
var router = express.Router();  
  
// Créer une instance de categorie.  
const Categorie = require('../models/categorie');  
  
// afficher la liste des categories.  
router.get('/', async (req, res, )=> {  
});  
  
// créer un nouvelle catégorie  
router.post('/', async (req, res) => {  
});  
  
// chercher une catégorie  
router.get('/:categorieId',async(req, res)=>{  
});  
  
// modifier une catégorie  
router.put('/:categorieId', async (req, res)=> {  
});  
  
// Supprimer une catégorie  
router.delete('/:categorieId', async (req, res)=> {  
});  
module.exports = router;
```

2. Méthode pour ajouter une catégorie. Modifier le contenu de routes/categorie.route.js

La propriété **req.body** contient des paires clé-valeur de données soumises dans le corps de la requête. Par défaut, il n'est pas défini et est rempli lorsque vous utilisez un middleware appelé body-parsing tel que `express.urlencoded()` ou `express.json()` (**voir app.js**).

La méthode **save()** renvoie une promesse. Si `save()` réussit, la promesse résout le document qui a été enregistré dans la base de donnée. On répond par le status 200 et le contenu json de la catégorie enregistrée.

Même si la base de données n'est pas préalablement créée, elle le sera suite à cette requête POST.

```
// créer une nouvelle catégorie
router.post('/', async (req, res) => {
  const { nomcategorie, imagecategorie } = req.body;
  const newCategorie = new Categorie({nomcategorie:nomcategorie,
imagecategorie:imagecategorie})
  try {
    await newCategorie.save();
    res.status(200).json(newCategorie );
  } catch (error) {
    res.status(404).json({ message: error.message });
  }
});
```

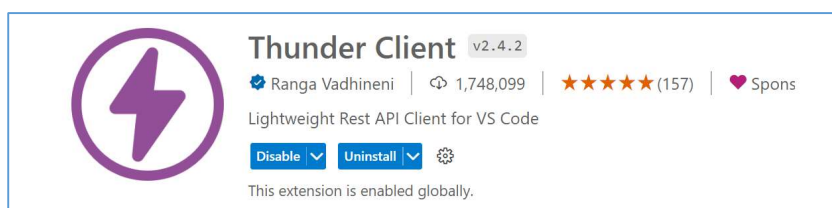
Dans le fichier **app.js** ajouter la route suivante :

```
const categorieRouter =require("./routes/categorie.route")
app.use('/api/categories', categorieRouter);
```

Attention : la ligne `app.use` devrait être placée après `const app = express();`

Tester le service créé :

Installer dans visual studio code l'extension **thunder client** qui est une extension client API Rest légère pour Visual Studio Code.



Démarrer thunder client pour tester l'ajout dans la base de données à travers l'url :

<http://localhost:3001/api/categories>

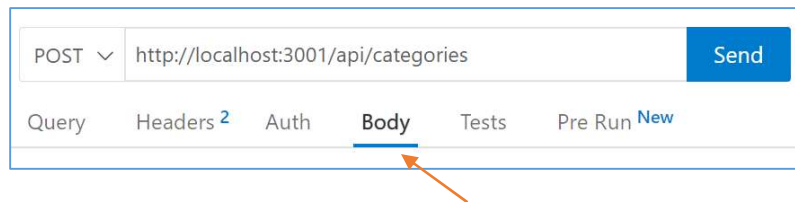
localhost : poste local

3001 : le numéro de port que nous avons choisi (*valeur dans .env*)

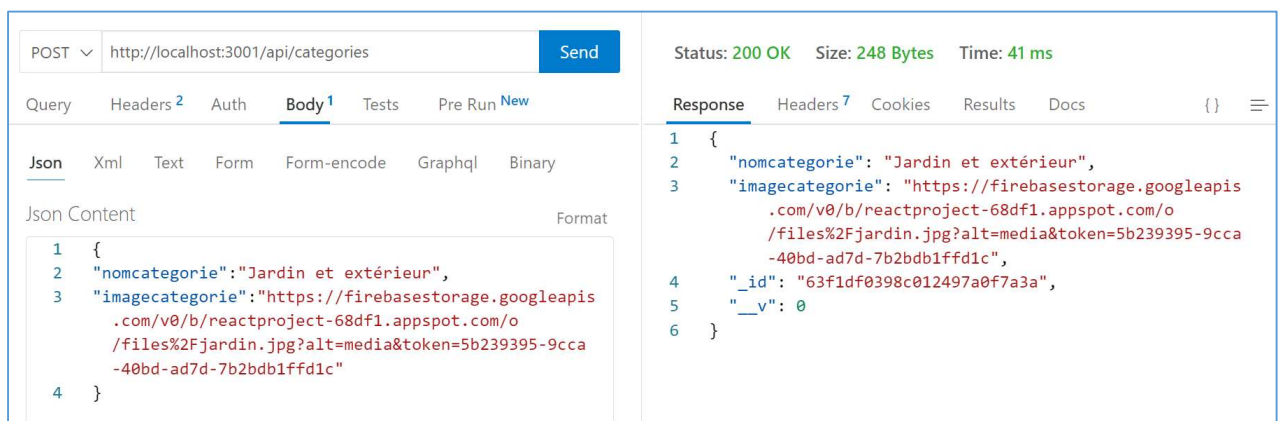
api/categories : la route spécifiée dans app.js : ***app.use('/api/categories', categorieRouter);*** en concaténation avec routes/categorie.route.js ***router.post('/', ...***

La requête pour ajouter est **post**

<http://localhost:3001/api/categories/>



Saisir le code json en respectant le schéma du model créé.



3. Méthode pour afficher la liste des catégories. Modifier le contenu de routes/categorie.route.js

La fonction **find()** est utilisée pour trouver des données particulières dans la base de données MongoDB. On recherche tous les documents dans la collection "categories", puis on renvoie tous leurs champs et les trie par `_id` en ordre décroissant.

```
const express = require('express');
const router = express.Router();
const Categorie=require("../models/categorie")

// afficher la liste des categories.
router.get('/', async (req, res )=> {
  try {
```

```

    const cat = await Categorie.find({}, null, {sort: {'_id': -1}})

    res.status(200).json(cat);
  } catch (error) {
    res.status(404).json({ message: error.message });
  }
});

```

await: Ce mot-clé est utilisé dans les fonctions asynchrones pour attendre que la promesse soit résolue avant de passer à la ligne suivante. Cela signifie que le code attendra que la requête à la base de données soit terminée avant de continuer.

Categorie: C'est le modèle Mongoose associé à une collection MongoDB. Dans ce cas, il s'agit du modèle pour la collection "categories".

find({}): Cette méthode recherche des documents dans la collection. Dans ce cas, {} spécifie qu'aucun critère de recherche n'est spécifié, ce qui signifie que tous les documents de la collection seront renvoyés.

null: Le deuxième argument est utilisé pour spécifier les champs à renvoyer, mais dans ce cas, il est défini sur null, ce qui signifie que tous les champs seront renvoyés.

{sort: {'_id': -1}}: Cet objet est utilisé pour spécifier le tri des résultats. Dans ce cas, le tri se fait sur le champ _id en ordre décroissant (-1), ce qui signifie que les documents seront triés du plus récent au plus ancien en fonction de leur _id.

Tester le service créé :

GET <http://localhost:3001/api/categories/>

The screenshot shows a REST client interface. The top bar indicates a GET request to <http://localhost:3001/api/categories/> with a status of 200 OK, size of 486 Bytes, and time of 33 ms. The left pane shows the 'Query' tab with 'Query Parameters' and a table with columns 'parameter' and 'value'. The right pane shows the 'Response' tab with a JSON array of two category objects. The first object has an '_id' of '63f1df0398c012497a0f7a3a', a 'nomcategorie' of 'Jardin et extérieur', and an 'imagecategorie' pointing to a Firebase storage URL. The second object has an '_id' of '63f1e01f9b6445fa4ec57c58', a 'nomcategorie' of 'bureaux', and an 'imagecategorie' pointing to another Firebase storage URL.

```

1  [
2    {
3      "_id": "63f1df0398c012497a0f7a3a",
4      "nomcategorie": "Jardin et extérieur",
5      "imagecategorie": "https://firebasestorage.googleapis
6      .com/v0/b/reactproject-68df1.appspot.com/o
7      /files%2Fjardin.jpg?alt=media&token=5b239395-9cca
8      -40bd-ad7d-7b2bdb1ffd1c",
9      "__v": 0
10   },
11   {
12     "_id": "63f1e01f9b6445fa4ec57c58",
13     "nomcategorie": "bureaux",
14     "imagecategorie": "https://firebasestorage.googleapis
15     .com/v0/b/reactproject-68df1.appspot.com/o
16     /files%2Fbureau.jpg?alt=media&token=fcbe6eae-84df
17     -4f1e-86bf-ac42751a01b4",
18     "__v": 0
19   }
20 ]

```

4. Méthode pour modifier une catégorie. Modifier le contenu de routes/categorie.route.js

La propriété **req.params** est un objet contenant des propriétés mappées aux "paramètres" de la route nommée. Par exemple, si vous avez la route `/api/categories/:categoryId`, la propriété « categoryId » est disponible en tant que `req.params.categoryId`. Cet objet est par défaut {}.

La fonction **findByIdAndUpdate()** est utilisée pour rechercher un document correspondant, le met à jour en fonction de l'argument de mise à jour, en transmettant toutes les options, et renvoie le document trouvé (le cas échéant) au rappel.

```
// modifier une catégorie
router.put('/:categoryId', async (req, res)=> {

  try {

    const cat1 = await Categorie.findByIdAndUpdate(
      req.params.categoryId,
      { $set: req.body },
      { new: true }
    );
    res.status(200).json(cat1);

  } catch (error) {
    res.status(404).json({ message: error.message });
  }
});
```

const cat1: Cela déclare une variable cat1 qui sera utilisée pour stocker le document mis à jour après l'exécution de la requête.

await: Comme dans l'exemple précédent, await est utilisé pour attendre la résolution de la promesse retournée par la méthode findByIdAndUpdate.

Categorie.findByIdAndUpdate: C'est la méthode Mongoose qui permet de trouver un document par son _id et de le mettre à jour. Elle prend généralement trois arguments : l'ID du document à mettre à jour, les données à mettre à jour (\$set: req.body dans ce cas), et des options (dans ce cas, { new: true }).

\$set est utilisé pour définir la valeur d'un champ dans un document MongoDB lors d'une opération de mise à jour, en le modifiant s'il existe déjà ou en l'ajoutant s'il n'existe pas.

req.params.categoryId: C'est l'ID de la catégorie à mettre à jour, extrait des paramètres de la requête HTTP. Il est typiquement fourni dans l'URL de la requête.

{ \$set: req.body }: C'est un objet utilisé pour spécifier les modifications à apporter au document. Dans cet exemple, il utilise l'opérateur \$set pour mettre à jour les champs spécifiés dans req.body. req.body contient généralement les données envoyées dans le corps de la requête HTTP, souvent lorsqu'un formulaire est soumis ou lorsqu'un client envoie des données JSON.

{ new: true }: C'est une option de configuration qui indique à Mongoose de renvoyer le document mis à jour plutôt que le document original avant la mise à jour. Cela signifie que cat1 contiendra le document mis à jour après l'exécution de la requête.

Il faut préciser la valeur de l'_id dans la requête PUT.

Tester le service créé :

PUT http://localhost:3001/api/categories/valeur_id

The screenshot shows a REST client interface with a PUT request to `http://localhost:3001/api/categories/63f1df0398c012497a0f7a3a`. The request body is a JSON object: `{ "nomcategorie": "Jardins et Extérieur", "imagecategorie": "https://firebasestorage.googleapis.com/v0/b/reactproject-68df1.appspot.com/o/files%2Fjardin.jpg?alt=media&token=5b239395-9cca-40bd-ad7d-7b2bdb1ffd1c" }`. The response is a 200 OK status with a JSON body: `{ "_id": "63f1df0398c012497a0f7a3a" }`.

5. Méthode pour supprimer une catégorie. Modifier le contenu de `routes/categorie.route.js`

La fonction **findByIdAndDelete()** est utilisée pour rechercher un document correspondant, le supprimer et transmettre le document trouvé (le cas échéant) au rappel.

```
// Supprimer une catégorie
router.delete('/:categorieId', async (req, res) => {
  const id = req.params.categorieId;
  await Categorie.findByIdAndDelete(id);
  res.json({ message: "categorie deleted successfully." });
});
```

Il faut préciser la valeur de l'_id dans la requête DELETE.

Tester le service créé :

DELETE http://localhost:3001/api/categories/valeur_id

The screenshot shows a REST client interface with a DELETE request to `http://localhost:3001/api/categories/63f1e1f0777afee7954`. The response is a 200 OK status with a JSON body: `{ "message": "categorie deleted successfully." }`.

6. Méthode pour chercher une catégorie

La fonction **findById()** est utilisée pour rechercher un seul document par son champ _id.

```
// chercher une catégorie
router.get('/:categorieId', async (req, res) => {
  try {
```

```

    const cat = await Categorie.findById(req.params.categorieId);

    res.status(200).json(cat);
  } catch (error) {
    res.status(404).json({ message: error.message });
  }
});

```

Il faut préciser la valeur de l'_id dans cette requête GET.

Tester le service créé :

GET http://localhost:3001/api/categories/valeur_id

The screenshot shows a REST client interface with the following details:

- Method:** GET
- URL:** http://localhost:3001/api/categories/63f1df0398c012497a0f7a3a
- Status:** 200 OK
- Size:** 249 Bytes
- Time:** 39 ms
- Response:**

```

1 {
2   "_id": "63f1df0398c012497a0f7a3a",
3   "nomcategorie": "Jardins et Extérieur",
4   "imagecategorie": "https://firebasestorage.googleapis.com/v0/b/reactproject-68df1.appspot.com/o/files%2Fjardin.jpg?alt=media&token=5b239395-9cca-40bd-ad7d-7b2bdb1ffd1c",
5   "_v": 0
6 }

```

Fichier categorie.route.js complet

```

var express = require('express');
var router = express.Router();

// Créer une instance de categorie.
const Categorie = require('../models/categorie');

// afficher la liste des categories.
router.get('/', async (req, res) => {
  try {
    const cat = await Categorie.find({}, null, {sort: {'_id': -1}})

    res.status(200).json(cat);
  } catch (error) {
    res.status(404).json({ message: error.message });
  }
});

// créer un nouvelle catégorie
router.post('/', async (req, res) => {

```

```

    const { nomcategorie, imagecategorie } = req.body;
    const newCategorie = new Categorie({nomcategorie:nomcategorie,
imagecategorie:imagecategorie})
    try {
        await newCategorie.save();
        res.status(200).json(newCategorie );
    } catch (error) {
        res.status(404).json({ message: error.message });
    }
});

// chercher une catégorie
router.get('/:categorieId', async (req, res) => {
    try {
        const cat = await Categorie.findById(req.params.categorieId);

        res.status(200).json(cat);
    } catch (error) {
        res.status(404).json({ message: error.message });
    }
});

// modifier une catégorie
router.put('/:categorieId', async (req, res) => {

    try {

        const cat1 = await Categorie.findByIdAndUpdate(
            req.params.categorieId,
            { $set: req.body },
            { new: true }
        );
        res.status(200).json(cat1);

    } catch (error) {
        res.status(404).json({ message: error.message });
    }
});

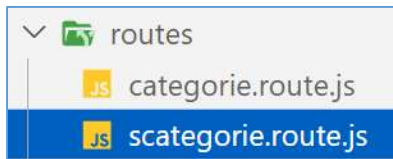
// Supprimer une catégorie
router.delete('/:categorieId', async (req, res) => {
    const id = req.params.categorieId;
    await Categorie.findByIdAndDelete(id);
    res.json({ message: "categorie deleted successfully." });
});

module.exports = router;

```

II. CRUD Sous categories

Créer le fichier routes/scategorie.route.js



Mongoose a la méthode **populate()**, qui vous permet de référencer des documents dans d'autres collections.

Le remplissage est le processus de remplacement automatique des chemins spécifiés dans le document par des documents d'autres collections.

La fonction exec() est utilisée pour exécuter la requête. Elle peut gérer les promesses et exécute facilement la requête. Le rappel peut être passé en tant que paramètre facultatif pour gérer les erreurs et les résultats.

```
const express = require('express');
const router = express.Router();
const SCategorie=require("../models/scategorie")

// afficher la liste des s/categories.
router.get('/', async (req, res, )=> {
  try {
    const scat = await SCategorie.find({}, null, {sort: {'_id': -1}}).populate("categorieID")

    res.status(200).json(scat);
  } catch (error) {
    res.status(404).json({ message: error.message });
  }
});

// créer une nouvelle s/catégorie
router.post('/', async (req, res) => {
  const { nomscategorie, imagescat,categorieID} = req.body;
  const newSCategorie = new SCategorie({nomscategorie:nomscategorie,
imagescat:imagescat,categorieID:categorieID })

  try {
    await newSCategorie.save();

    res.status(200).json(newSCategorie );
  } catch (error) {
    res.status(404).json({ message: error.message });
  }
});
```

```

});

// chercher une sous catégorie
router.get('/:scategorieId', async (req, res) => {
  try {
    const scat = await SCategorie.findById(req.params.scategorieId);

    res.status(200).json(scat);
  } catch (error) {
    res.status(404).json({ message: error.message });
  }
});

// modifier une s/catégorie
router.put('/:scategorieId', async (req, res) => {

  try {

    const scat1 = await SCategorie.findByIdAndUpdate(
      req.params.scategorieId,
      { $set: req.body },
      { new: true }
    );
    res.status(200).json(scat1);

  } catch (error) {
    res.status(404).json({ message: error.message });
  }
});

// Supprimer une s/catégorie
router.delete('/:scategorieId', async (req, res) => {
  const id = req.params.scategorieId;
  await SCategorie.findByIdAndDelete(id);

  res.json({ message: "sous categorie deleted successfully." });
});

// chercher une sous catégorie par cat
router.get('/cat/:categorieID', async (req, res) => {
  try {
    const scat = await SCategorie.find({ categorieID:
req.params.categorieID }).exec();

    res.status(200).json(scat);
  } catch (error) {
    res.status(404).json({ message: error.message });
  }
});

```



```
module.exports = router;
```

Ajouter la route du fichier dans **app.js**

JS app.js >

```
const categorieRouter = require("./routes/scategorie.route")  
  
app.use('/api/scategories', categorieRouter);
```

Tester les CRUDs en utilisant **thunder client**

Attention : Tâcher de donner des valeurs existantes dans categories pour categorieID dans la requête POST de scategories.

Exemple de résultat :

The screenshot shows the Thunder Client interface. On the left, a GET request is configured to `http://localhost:3001/api/scategories/`. The 'Query Parameters' section is empty. On the right, the response is displayed as a JSON array with three category objects. The status is 200 OK, size is 1.29 KB, and time is 11 ms.

```
1 [
2   {
3     "_id": "63f1e8516131d372ffbbd753",
4     "nomscategorie": "Meubles Extérieur",
5     "imagescat": "https://www.sauvaje.fr/wp-content/uploads/2017/08/DSC_0636-1600x1063.jpg",
6     "categorieID": {
7       "_id": "63f1df0398c012497a0f7a3a",
8       "nomcategorie": "Jardins et Extérieur",
9       "imagecategorie": "https://firebasestorage.googleapis.com/v0/b/reactproject-68df1.appspot.com/o/files%2Fjardin.jpg?alt=media&token=5b239395-9cca-40bd-ad7d-7b2bdb1ffdc",
10      "__v": 0
11    },
12    "__v": 0
13  },
14  {
15    "_id": "63f1e8896131d372ffbbd755",
16    "nomscategorie": "Chaises en bois",
17    "imagescat": "https://jardipartage.b-cdn.net/wp-content/uploads/2021/08/conseils-pour-reussir-la-decoration-de-son-balcon-780x470.jpg",
18    "categorieID": {
19      "_id": "63f1df0398c012497a0f7a3a",
20      "nomcategorie": "Jardins et Extérieur",
21      "imagecategorie": "https://firebasestorage.googleapis.com/v0/b"
```

On voit bien le contenu généré de categories grâce à populate dans la requête GET.

III. CRUD articles

Créer le fichier `routes/article.route.js`



```
const express = require('express');
const router = express.Router();
const Article=require("../models/article")

// afficher la liste des articles.
router.get('/', async (req, res, )=> {
  try {
    const articles = await Article.find({}, null, {sort: {'_id': -1}}).populate("scategorieID").exec();

    res.status(200).json(articles);
  } catch (error) {
    res.status(404).json({ message: error.message });
  }
});

// créer un nouvel article
router.post('/', async (req, res) => {

  const nouvarticle = new Article(req.body)
  try {
    const response =await nouvarticle.save();
    const articles = await
Article.findById(response._id).populate("scategorieID").exec();
    res.status(200).json(articles);
  } catch (error) {
    res.status(404).json({ message: error.message });
  }
});

// chercher un article
router.get('/:articleId',async(req, res)=>{
  try {
    const art = await Article.findById(req.params.articleId);

    res.status(200).json(art);
  } catch (error) {
    res.status(404).json({ message: error.message });
  }
});

// modifier un article
```

```

router.put('/:articleId', async (req, res)=> {
  try {
    const art = await Article.findByIdAndUpdate(
      req.params.articleId,
      { $set: req.body },
      { new: true }
    );
    const articles = await
Article.findById(art._id).populate("scategorieID").exec();
    res.status(200).json(articles);
  } catch (error) {
    res.status(404).json({ message: error.message });
  }
});
// Supprimer un article
router.delete('/:articleId', async (req, res)=> {
  const id = req.params.articleId;
  try {
    await Article.findByIdAndDelete(id);

    res.status(200).json({ message: "article deleted successfully." });
  } catch (error) {
    res.status(404).json({ message: error.message });
  }
});
module.exports = router;

```

Dans le fichier **app.js** ajouter la route de l'article.

 app.js >

```

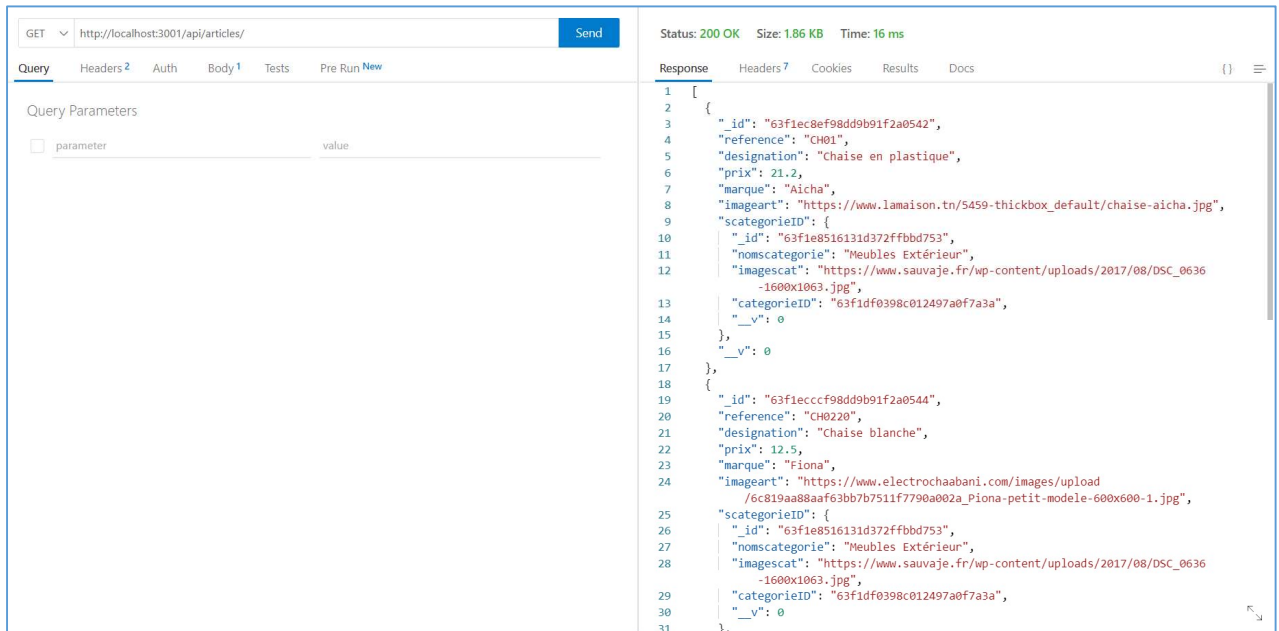
const articleRouter =require("./routes/article.route")

app.use('/api/articles', articleRouter);

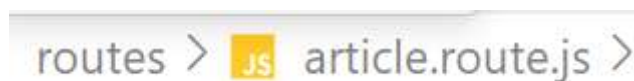
```

Tester les CRUDs de l'article en utilisant thunder client en donnant des valeurs existantes dans scategories pour scategorieID dans la requête POST.

Exemple de résultat où on voit bien le contenu généré de scategories grâce à populate dans la requête GET.



Par la suite on va ajouter des requêtes supplémentaires qu'on va intégrer à **article.route.js**



On veut afficher la liste des articles pour une sous catégorie donnée

```

// chercher un article par s/cat
router.get('/scat/:scategorieID', async (req, res) => {
  try {
    const art = await Article.find({ scategorieID:
req.params.scategorieID }).exec();

    res.status(200).json(art);
  } catch (error) {
    res.status(404).json({ message: error.message });
  }
});

```

GET <http://localhost:3001/api/articles/scat/63f1e8516131d372ff...> Send

Status: 200 OK Size: 1.14 KB Time: 16 ms

Query Parameters

Response

```

1  [
2    {
3      "_id": "63f1ec8ef98dd9b91f2a0542",
4      "reference": "CH012",
5      "designation": "Chaise en plastique",
6      "prix": 201.2,
7      "marque": "Aicha",
8      "imageart": "https://www.lamaison.tn/5459
          -thickbox_default/chaise-aicha.jpg",
9      "scategorieID": "63f1e8516131d372ffbbd753",
10     "__v": 0,
11     "qttestock": 5
12   },
13   {
14     "_id": "63f1eccc98dd9b91f2a0544",
15     "reference": "CH0220",
16     "designation": "Chaise blanche",
17     "prix": 12.5,

```

On veut afficher la liste des articles pour une catégorie donnée

```

const Scategorie = require("../models/scategorie")

// chercher un article par cat
router.get('/cat/:categorieID', async (req, res) => {
  try {
    // Recherche des sous-catégories correspondant à la catégorie donnée
    const sousCategories = await Scategorie.find({ categorieID:
req.params.categorieID }).exec();

    // Initialiser un tableau pour stocker les identifiants des sous-
catégories trouvées
    const sousCategorieIDs = sousCategories.map(scategorie => scategorie._id);

    // Recherche des articles correspondant aux sous-catégories trouvées
    const articles = await Article.find({ scategorieID: { $in:
sousCategorieIDs } }).exec();

    res.status(200).json(articles);
  } catch (error) {
    res.status(404).json({ message: error.message });
  }
});

```

GET <http://localhost:3001/api/articles/cat/63f1df0398c012497a0> Send

Status: 200 OK Size: 1.83 KB Time: 10 ms

Query Headers 2 Auth Body 1 Tests Pre Run

Response Headers 7 Cookies Results Docs

Query Parameters

parameter value

Response

```

1 [
2   {
3     "_id": "63f1ec8ef98dd9b91f2a0542",
4     "reference": "CH012",
5     "designation": "Chaise en plastique",
6     "prix": 201.2,
7     "marque": "Aicha",
8     "imageart": "https://www.lamaison.tn/5459
    -thickbox_default/chaise-aicha.jpg",
9     "scategorieID": "63f1e8516131d372ffb753",
10    "_v": 0,
11    "qtestock": 5
12  },
13  {
14    "_id": "63f1eccccf98dd9b91f2a0544",
15    "reference": "CH0220",
16    "designation": "Chaise blanche",
17    "prix": 12.5,

```

On veut afficher la liste des articles pour une page et une limite données de nombres d'enregistrements et pour un numéro de page donné.

Attention : Il faut placer cette requête avant les routes qui attendent un ObjectId. Avec cette organisation, Express.js correspondra d'abord à la route de pagination si l'URL correspond à /api/articles/pagination, évitant ainsi le conflit avec la route attendue pour un identifiant ObjectId. Autrement la route /api/articles/pagination entrera en conflit avec une route qui attend un ObjectId dans le chemin. Cela va arriver la route /api/articles/:id où :id est supposé être un identifiant ObjectId. Ce qui va engendrer l'erreur "Cast to ObjectId failed" qui indique qu'il y a une tentative de conversion d'une valeur qui n'est pas un ObjectId en un ObjectId.

```

// afficher la liste des articles par page
router.get('/pagination', async(req, res) => {
  const page = req.query.page || 1 // Current page
  const limit = req.query.limit || 5; // Number of items per page

  // Calculez le nombre d'éléments à sauter (offset)
  const offset = (page - 1) * limit;

  try {

    // Effectuez la requête à votre source de données en utilisant les paramètres
    de pagination

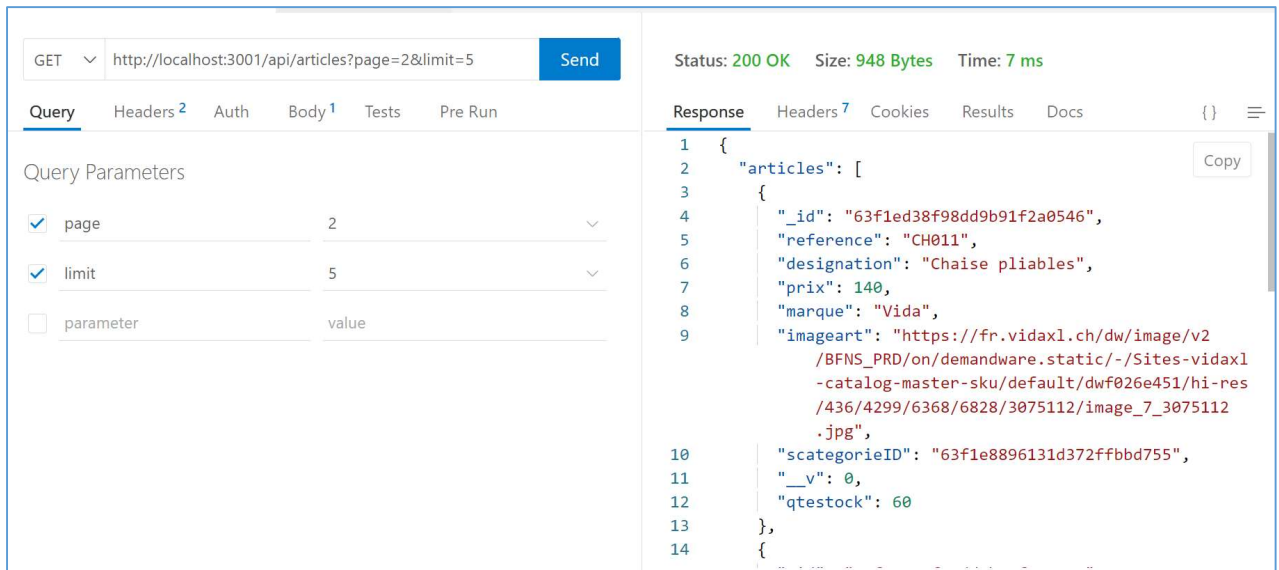
    const articlesTot = await Article.countDocuments();

    const articles = await Article.find( {}, null, {sort: {'_id': -1}})
      .skip(offset)
      .limit(limit)

    res.status(200).json({articles:articles,tot:articlesTot});
  } catch (error) {
    res.status(404).json({ message: error.message });
  }
});

```

http://localhost:3001/api/articles?page=2&limit=5



GET Send

Status: 200 OK Size: 948 Bytes Time: 7 ms

Query Parameters

<input checked="" type="checkbox"/>	page	2
<input checked="" type="checkbox"/>	limit	5
<input type="checkbox"/>	parameter	value

Response

```
1 {
2   "articles": [
3     {
4       "_id": "63f1ed38f98dd9b91f2a0546",
5       "reference": "CH011",
6       "designation": "Chaise pliables",
7       "prix": 140,
8       "marque": "Vida",
9       "imageart": "https://fr.vidaxl.ch/dw/image/v2
        /BFNS_PRD/on/demandware.static/-/Sites-vidaxl
        -catalog-master-sku/default/dwf026e451/hi-res
        /436/4299/6368/6828/3075112/image_7_3075112
        .jpg",
10      "scategorieID": "63f1e8896131d372ffbbd755",
11      "__v": 0,
12      "qtestock": 60
13    },
14    {
```

Le code complet de /routes/article.route.js deviendra :

```
const express = require('express');
const router = express.Router();
const Article=require("../models/article")
const Scategorie =require("../models/scategorie")

// afficher la liste des articles.

router.get('/', async (req, res, )=> {
  try {
    const articles = await Article.find({}, null, {sort: {'_id': -1}}).populate("scategorieID").exec();

    res.status(200).json(articles);
  } catch (error) {
    res.status(404).json({ message: error.message });
  }
});

// créer un nouvel article
router.post('/', async (req, res) => {

  const nouvarticle = new Article(req.body)

  try {
    await nouvarticle.save();

    res.status(200).json(nouvarticle );
  } catch (error) {
    res.status(404).json({ message: error.message });
  }
});
```

```

    }

});

// afficher la liste des articles par page
router.get('/pagination', async(req, res) => {
    const page = req.query.page || 1 // Current page
    const limit = req.query.limit || 5; // Number of items per page

    // Calculez le nombre d'éléments à sauter (offset)
    const offset = (page - 1) * limit;

    try {

        // Effectuez la requête à votre source de données en utilisant les paramètres
        // de pagination

        const articlesTot = await Article.countDocuments();

        const articles = await Article.find( {}, null, {sort: {'_id': -1}})
            .skip(offset)
            .limit(limit)

        res.status(200).json({articles:articles,tot:articlesTot});
    } catch (error) {
        res.status(404).json({ message: error.message });
    }
});

// chercher un article
router.get('/:articleId', async(req, res)=>{
    try {
        const art = await Article.findById(req.params.articleId);

        res.status(200).json(art);
    } catch (error) {
        res.status(404).json({ message: error.message });
    }
});

// modifier un article
router.put('/:articleId', async (req, res)=> {
    try {
        const art = await Article.findByIdAndUpdate(
            req.params.articleId,
            { $set: req.body },
            { new: true }
        );
        const articles = await
Article.findById(art._id).populate("scategorieID").exec();

```



```

        res.status(200).json(articles);
    } catch (error) {
        res.status(404).json({ message: error.message });
    }
});

// Supprimer un article
router.delete('/:articleId', async (req, res)=> {
    const id = req.params.articleId;
    await Article.findByIdAndDelete(id);

    res.json({ message: "article deleted successfully." });
});

// chercher un article par s/cat
router.get('/scat/:scategorieID', async (req, res)=>{
    try {
        const art = await Article.find({ scategorieID:
req.params.scategorieID}).exec();

        res.status(200).json(art);
    } catch (error) {
        res.status(404).json({ message: error.message });
    }
});

// chercher un article par cat
router.get('/cat/:categorieID', async (req, res) => {
    try {
        // Recherche des sous-catégories correspondant à la catégorie donnée
        const sousCategories = await Scategorie.find({ categorieID:
req.params.categorieID }).exec();

        // Initialiser un tableau pour stocker les identifiants des sous-
catégories trouvées
        const sousCategorieIDs = sousCategories.map(scategorie => scategorie._id);

        // Recherche des articles correspondant aux sous-catégories trouvées
        const articles = await Article.find({ scategorieID: { $in:
sousCategorieIDs } }).exec();

        res.status(200).json(articles);
    } catch (error) {
        res.status(404).json({ message: error.message });
    }
});

module.exports = router;

```