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.

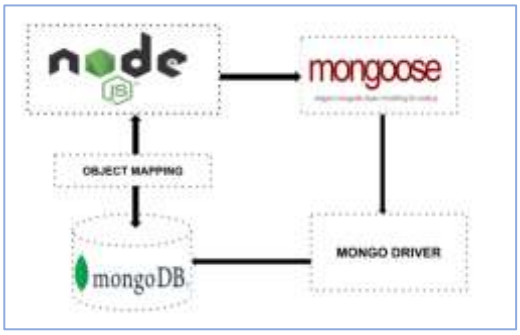
D'autre part, **MongoDB** est un système de gestion de base de données orienté documents, de la classe des bases de données NoSQL.

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 (facultatif)

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

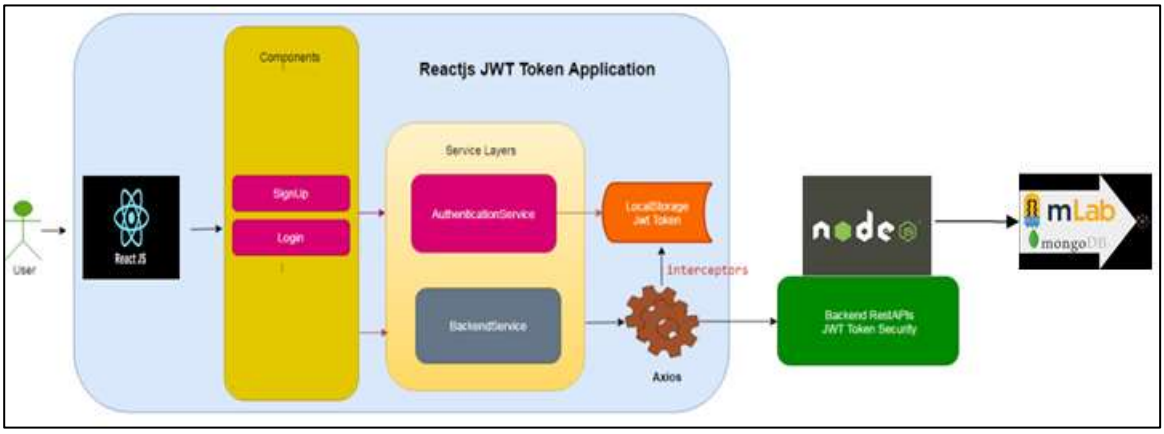
Etude ce cas : Site de Commerce en ligne

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 messager qui envoie votre demande au fournisseur, puis vous renvoie la réponse.

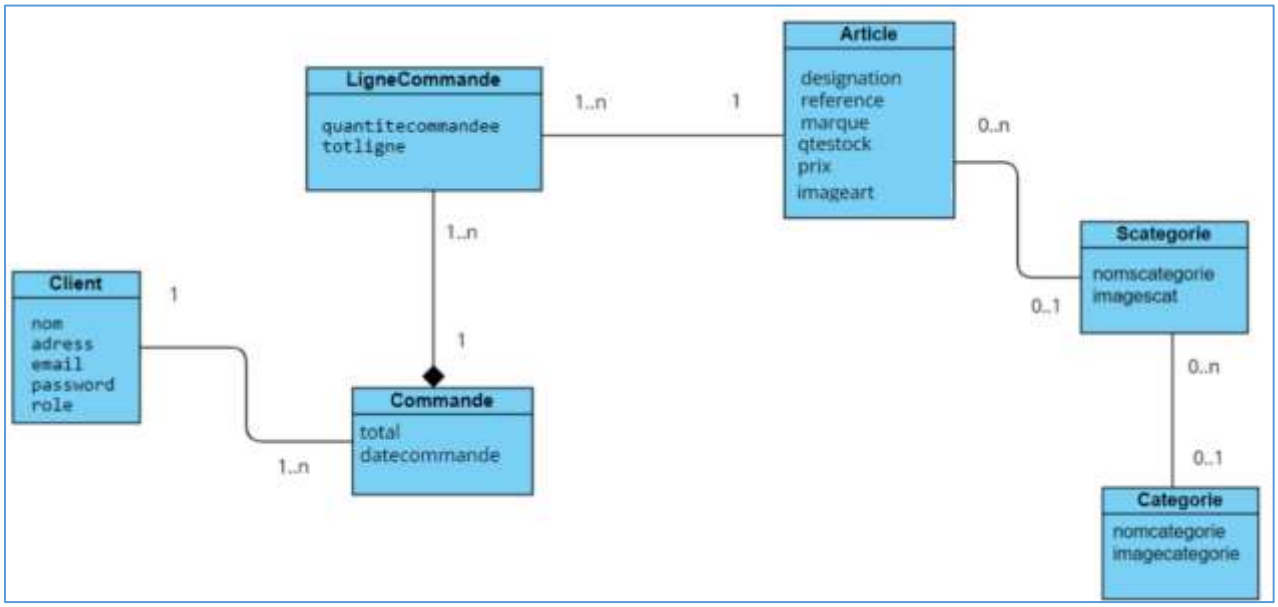
On va intégrer également la technique de JWT pour l'opération d'authentification.

JSON Web Token est un standard ouvert défini dans la RFC 7519. Il permet l'échange sécurisé de jetons entre plusieurs parties. Cette sécurité de l'échange se traduit par la vérification de l'intégrité des données à l'aide d'une signature numérique.

L'architecture de l'application :



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.

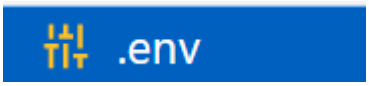
dotenv nous facilitera l'utilisation d'un fichier .env pour stocker des variables sensibles telles que le port ou le nom de la base de données.

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.

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




```
ENV=DEVELOPMENT
DATABASE=mongodb://127.0.0.1:27017/DatabaseCommerce
DATABASECLOUD=mongodb+srv://hassan:messk2015@cluster0.yfhhu.mongodb.net/DatabaseCommerce?retryWrites=true&w=majority
PORT=3001
```

5. Mettre le code suivant dans un nouveau fichier appelé app.js :

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

dotenv.config()
const app = express();

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

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

mongoose.set("strictQuery", false);
```

```
// Connexion à la base données
mongoose.connect(process.env.DATABASE,{
 useNewUrlParser: true,
 useUnifiedTopology: true
 })
 .then(() => {console.log("Connexion à la base de données réussie");
}).catch(err => {
 console.log('Impossible de se connecter à la base de données', err);
 process.exit();
});

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()**. Lorsque l'option strict est définie sur true (`strictQuery`), Mongoose s'assurera que seuls les champs spécifiés dans votre schéma seront enregistrés dans la base de données et que tous les autres champs ne seront pas enregistrés (si d'autres champs sont envoyés). À l'heure actuelle, cette option est activée par défaut, mais elle sera modifiée dans Mongoose v7 en false par défaut. Cela signifie que tous les champs seront enregistrés dans la base de données, même si certains d'entre eux ne sont pas spécifiés dans le modèle de schéma.

Donc, si vous voulez avoir des schémas stricts et stocker dans la base de données uniquement ce qui est spécifié dans votre modèle, à partir de Mongoose v7, vous devrez définir l'option stricte sur true manuellement.

unifiedtopology : DeprecationWarning : le moteur actuel de détection et de surveillance des serveurs est obsolète et sera supprimé dans une future version. Pour utiliser le nouveau moteur de découverte et de surveillance de serveur, transmettez l'option { useUnifiedTopology: true } au constructeur MongoClient.

usenewurlparser : DeprecationWarning : l'analyseur de chaîne d'URL actuel est obsolète et sera supprimé dans une future version. Pour utiliser le nouvel analyseur, transmettez l'option { useNewUrlParser: true } à MongoClient.connect.

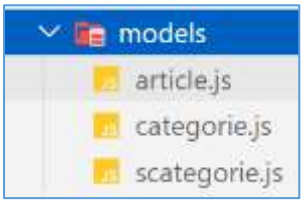
La fonction **app.listen()** est utilisée pour lier et écouter les connexions sur l'hôte et le port spécifiés. Si le numéro de port est omis ou est égal à 0, le système d'exploitation attribuera un port inutilisé arbitraire.

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. Pour tous les autres chemins d'accès, elle répondra par 404 Not Found.

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
Connexion à la base de données réussie
```

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 **ObjetId**, et avec l'option **ref**, nous avons dit à mangoose 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** :

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 });
    }
});
```

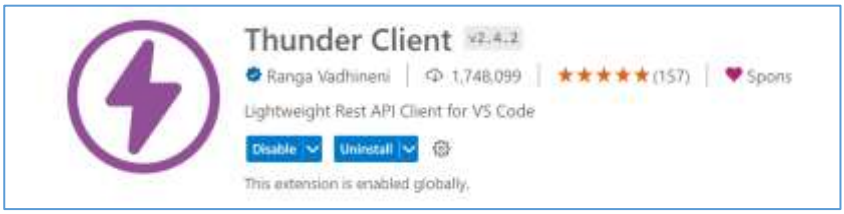
Dans le fichier **app.js** ajouter la route suivante :

```
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 :

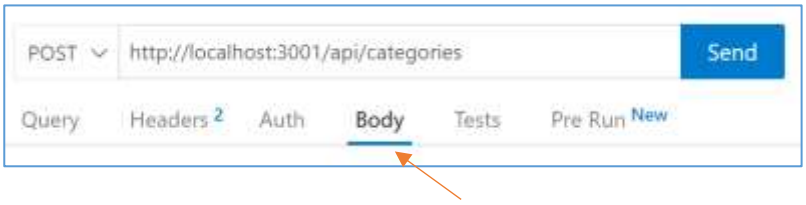
http://localhost:3001/api/categories

localhost : poste local

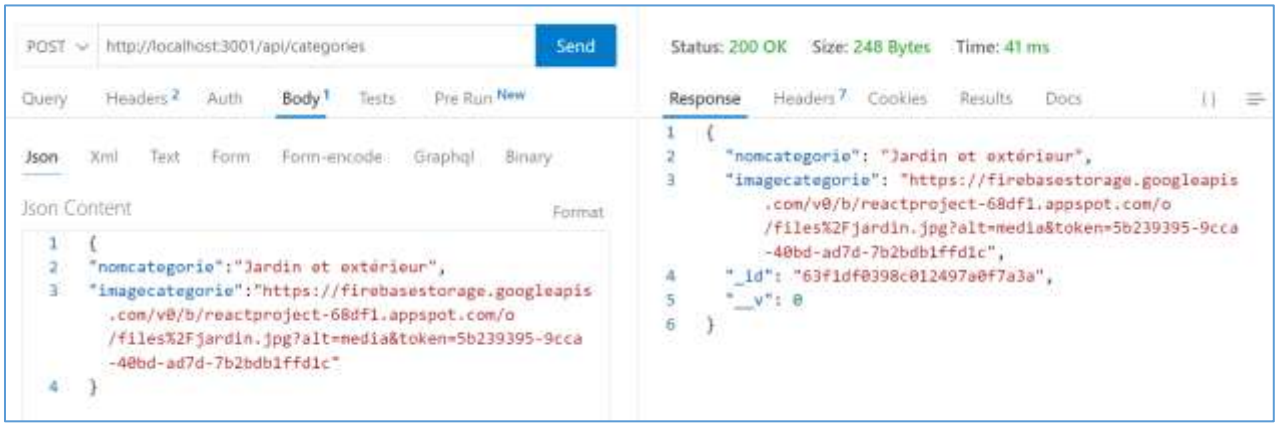
3001 : le numéro de port que nous avons choisi (***valeur dans .env***)

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***



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



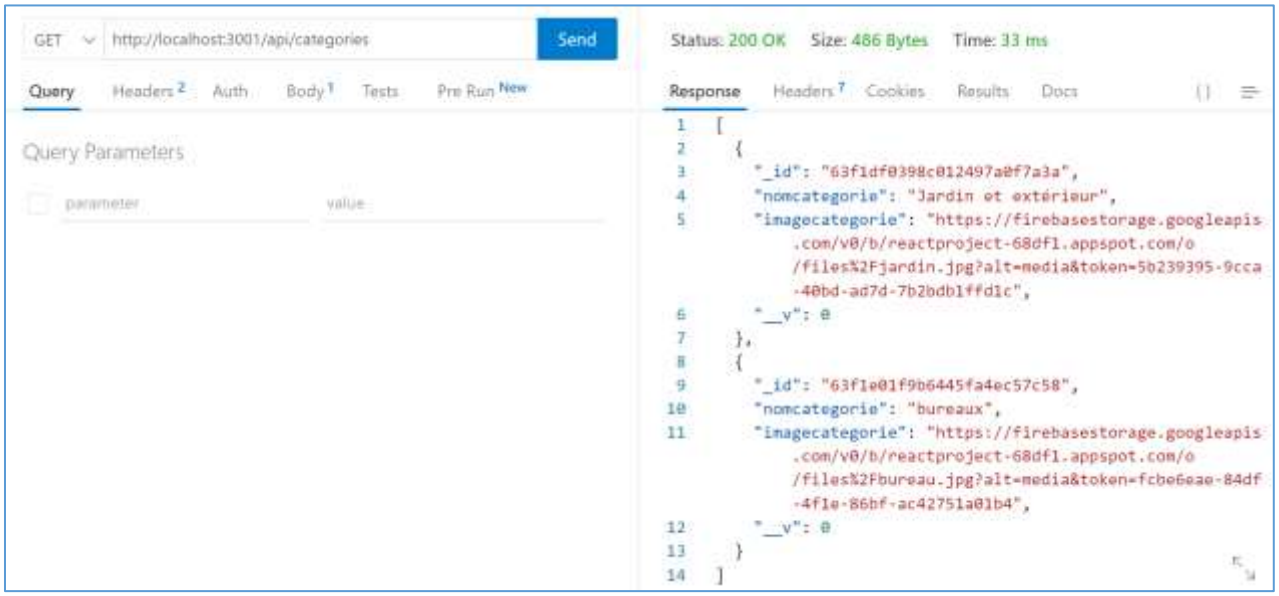
3. Méthode pour afficher la liste des catégories. Modifier le contenu de routes/categorie.route.js

La fonction **find()** est utilisée pour trouver des données particulières dans la base de données MongoDB.

```
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();

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



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/:categorieId***, la propriété « categorieId » est disponible en tant que req.params.categorieId. 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.

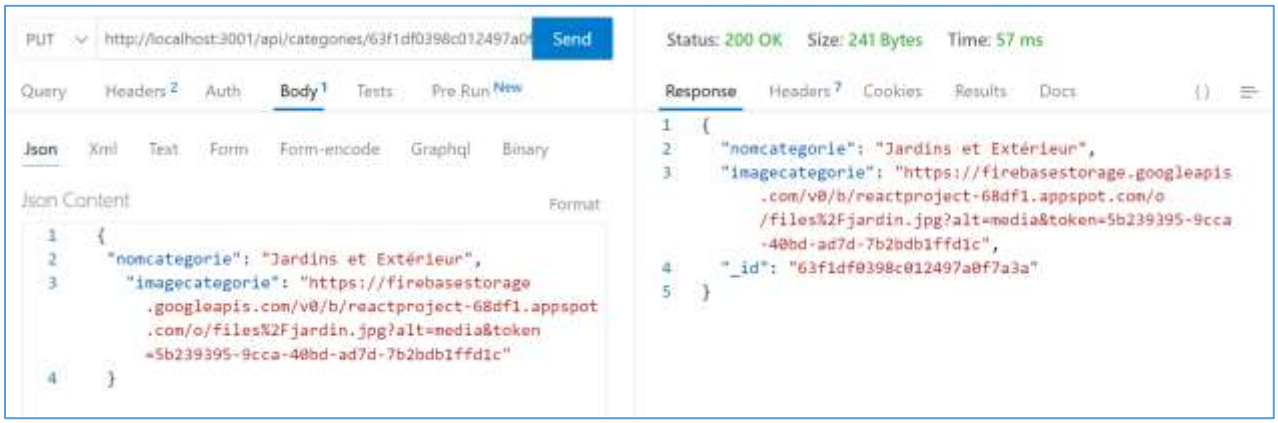
```
// modifier une catégorie
router.put('/:categorieId', async (req, res)=> {
    const { nomcategorie, imagecategorie} = req.body;
    const id  = req.params.categorieId;

    try {

    const cat1 = {
nomcategorie:nomcategorie,imagecategorie:imagecategorie, _id:id };
console.log(cat1)
    await Categorie.findByIdAndUpdate(id, cat1);

    res.json(cat1);
    } catch (error) {
    res.status(404).json({ message: error.message });
    }
});
```

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



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.

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 la requête DELETE.

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();

        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)=> {
    const { nomcategorie, imagecategorie} = req.body;
    const id  = req.params.categorieId;

    try {

    const cat1 = { nomcategorie:nomcategorie,imagecategorie:imagecategorie, _id:id
};
console.log(cat1)
    await Categorie.findByIdAndUpdate(id, cat1);

    res.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 categories.
router.get('/', async (req, res, )=> {
    try {
        const scat = await SCategorie.find().populate("categorieID").exec();

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

});

// créer un nouvelle 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 catégorie
router.put('/:scategorieId', async (req, res)=> {
    const { nomscategorie, imagescat,categorieID} = req.body;
    const id  = req.params.scategorieId;

    try {

    const scat1 = {
nomscategorie:nomscategorie,imagescat:imagescat,categorieID:categorieID, _id:id };

    await SCategorie.findByIdAndUpdate(id, scat1);

    res.json(scat1);
    } catch (error) {
    res.status(404).json({ message: error.message });
    }
});

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

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

});

module.exports = router;
```

Ajouter la route du fichier dans app.js

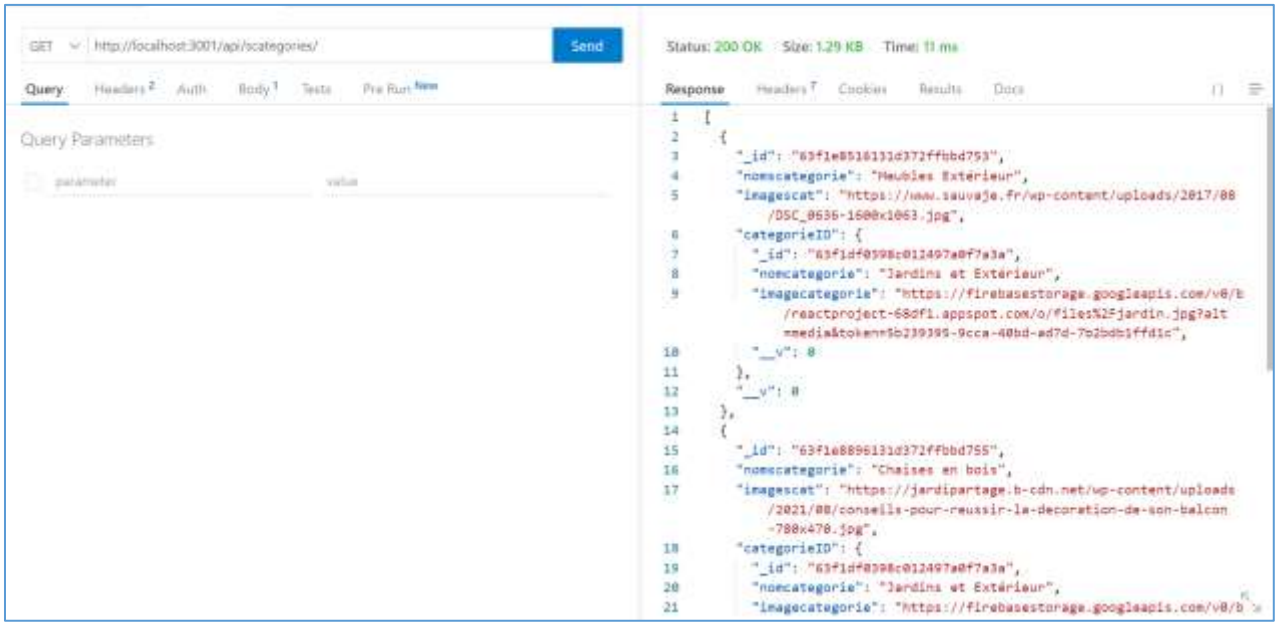
```
const scategorieRouter =require("./routes/scategorie.route")

app.use('/api/scategories', scategorieRouter);
```

Tester les CRUDs en utilisant **thunder client**

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

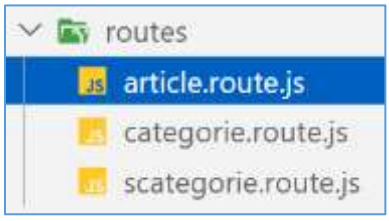
Exemple de résultat :



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().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 });
    }


});
// 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)=> {
    const { reference,
designation,prix,marque,qtestock,imageart,scategorieID} = req.body;
    const id  = req.params.articleId;

    try {

    const art1 = {
reference:reference,designation:designation,prix:prix,marque:marque,qte
stock:qtestock,imageart:imageart,scategorieID:scategorieID, _id:id };

    await Article.findByIdAndUpdate(id, art1);

    res.json(art1);
```

```
    } 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." });

});
module.exports = router;
```

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

```
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.