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.

Prisma est un ORM (Object Relational Mapper) open source pour Node.js. Il prend en charge à la fois JavaScript et TypeScript.

Les schémas aident les développeurs à éviter les problèmes d'incohérence des données au fil du temps. Bien que vous puissiez définir un schéma au niveau de la base de données dans MongoDB, Prisma vous permet de définir un schéma au niveau de l'application. Lorsqu'il utilise le client Prisma, un développeur bénéficie de l'aide de requêtes à saisie semi-automatique, puisque le client Prisma connaît le schéma.

En général, les données auxquelles on accède ensemble doivent être stockées ensemble dans une base de données MongoDB. Prisma prend en charge l'utilisation de documents intégrés pour conserver les données ensemble.

Cependant, il peut y avoir des cas d'utilisation dans lesquels vous devrez stocker les données associées dans des collections distinctes. Pour ce faire dans MongoDB, vous pouvez inclure le champ _id d'un document dans un autre document. Dans ce cas, Prisma peut vous aider à organiser ces données associées et à maintenir l'intégrité référentielle des données.

Prérequis :

Installer NodeJs

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

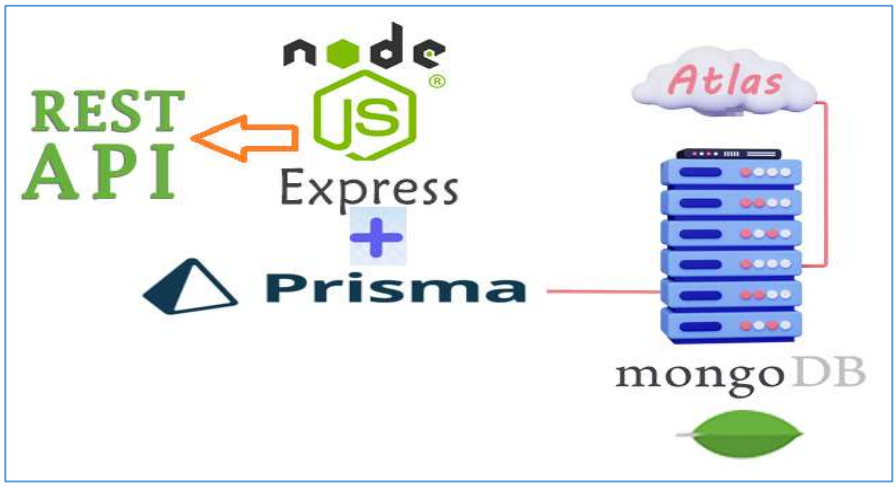
Créer un compte Atlas

Dans cet atelier, nous utiliserons un cluster MongoDB Atlas. Pour créer un compte gratuit et votre premier cluster définitivement gratuit allez au site :

https://www.mongodb.com/docs/atlas/getting-started/

Etude ce cas : Site de Commerce en ligne

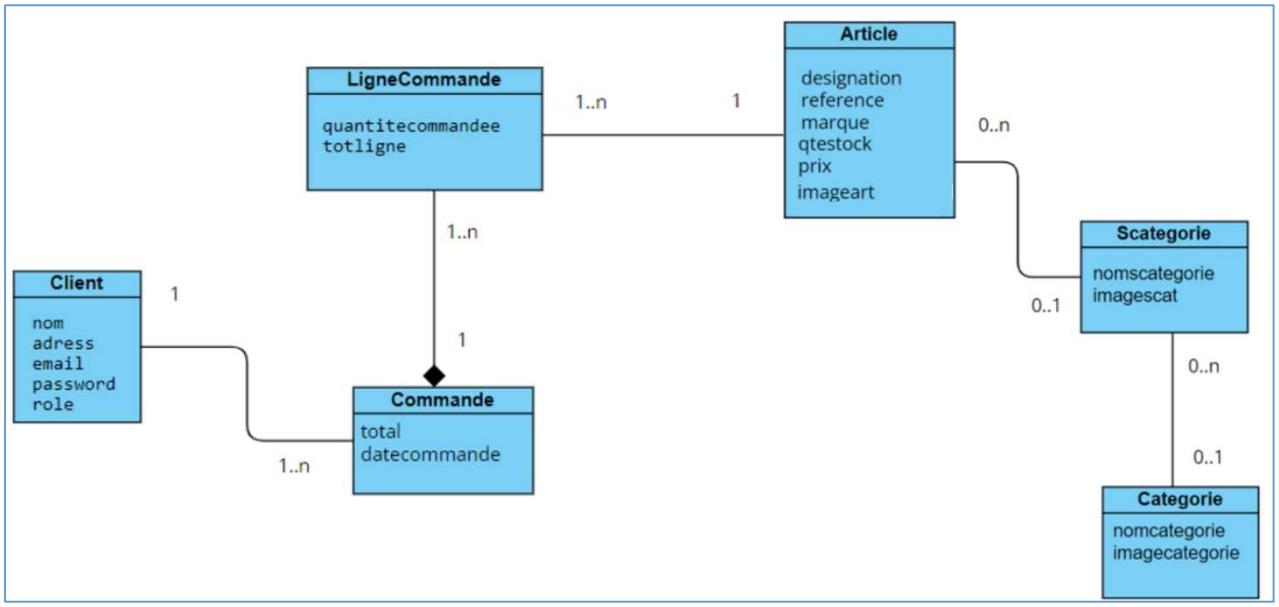
Notre objectif est de voir comment construire et créer une API CRUD REST en utilisant Node.js Express.js et MongoDB avec Prisma. Une API est une interface logicielle qui permet à deux applications de communiquer entre elles. En d'autres termes, une API est un messager qui envoie votre demande au fournisseur,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



.

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 prisma 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 et prisma.

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.

Prisma 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.

Créez maintenant votre configuration Prisma initiale à l'aide de la commande init de la CLI Prisma :

```
npx prisma init
```

Cette commande crée un nouveau répertoire prisma avec le contenu suivant :

schema.prisma : Spécifie votre connexion à la base de données et contient le schéma de la base de données

.env : un fichier dotenv, généralement utilisé pour stocker les informations d'identification de votre base de données dans un groupe de variables d'environnement

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

Supprimer la ligne

```
DATABASE_URL="postgresql://johndoe:randompassword@localhost:5432/mydb?schema=public"
```

Puis la remplacer par :

```
DATABASE_URL="[redacted]"
```

Remarque : remplacer la chaîne par celle donnée par atlas.

Puis ajouter :

```
PORT=3001
```

5. Ajouter ce code au fichier /prisma/schema.prisma


```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "mongodb"
  url      = env("DATABASE_URL")
}
```

```
model articles {
  id          String      @id @default(auto()) @map("_id") @db.ObjectId
  designation  String      @unique(map: "designation_1")
  imageart     String
  marque       String
  prix         Float
  qtestock     Int
  reference    String      @unique(map: "reference_1")
  scategorieID String      @db.ObjectId
  scategories  scategories  @relation(fields: [scategorieID], references: [id])
}

model categories {
  id            String      @id @default(auto()) @map("_id") @db.ObjectId
  imagecategorie String
  nomcategorie   String      @unique(map: "nomcategorie_1")
  scategories    scategories[]
}

model scategories {
  id            String      @id @default(auto()) @map("_id") @db.ObjectId
  imagescategorie String
  nomscategorie   String
  categorieID     String
  categories      categories  @relation(fields: [categorieID], references: [id])
  articles        articles[]
}
```

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

```
const express=require('express');
const app = express()

//CORS
const cors = require('cors')
app.use(cors());

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

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


const PORT = process.env.PORT || 3001

app.listen(PORT, () => console.log(`Server is running on port ${PORT} `))
```

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.

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.

Exécuter le code avec la commande

nodemon app

```
[nodemon] starting `node app.js`
Server is running on port 3001
```

I. CRUD categories

1. On va maintenant créer les routes pour le cas **categorie** :

routes/categories.route.js

```
const express = require('express');
const { PrismaClient } = require('@prisma/client')
const prisma = new PrismaClient()
const router = express.Router();

//Ajout d'une catégorie
router.post('/', async (req, res, )=> {
    const {nomcategorie,imagecategorie}=req.body
    try {
        const categorie = await prisma.categories.create({
            data: {
                nomcategorie: nomcategorie,
                imagecategorie: imagecategorie
            }
        })

        res.json(categorie)
      } catch (error) {
        res.status(500).json({ message: error.message });
      }

});

// afficher la liste des catégories.
router.get('/', async (req, res, )=> {
    try {
        // categories est le nom du model précisé dans prisma.schema
        const categories = await prisma.categories.findMany()
         res.json(categories)
        } catch (error) {
         res.status(500).json({ message: error.message });
       }
});

// afficher une catégorie.
router.get('/:id', async (req, res, )=> {
    const { id } = req.params

    try {
        const categorie = await prisma.categories.findUnique({
            where: {
                id: id,
            }
```

```
        })

       res.json(categorie)
      } catch (error) {
        res.status(500).json({ message: error.message });
      }

});



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

    try {
     const categorie = await prisma.categories.update({
        data: {
            nomcategorie: nomcategorie,
            imagecategorie: imagecategorie
        },
            where: { id: id},
        })
    res.json(categorie);
    } catch (error) {
    res.status(404).json({ message: error.message });
    }
});

// Supprimer une catégorie
router.delete('/:id', async (req, res)=> {
    const  id  = req.params.id;
    try {
     await prisma.categories.delete({
        where: { id: id },
    })

    res.json({ message: "category "+ id +" deleted successfully." });
} catch (error) {
    res.status(404).json({ message: error.message });
    }
});

module.exports = router;
```

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 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 {}.

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

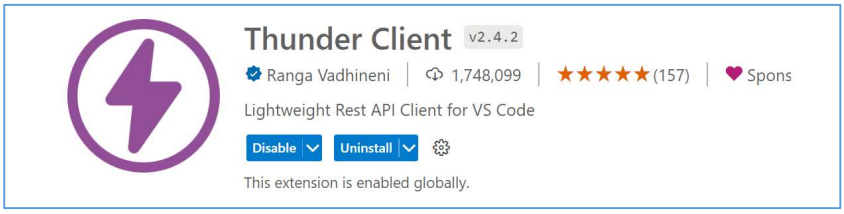
app.js

```
// Appel de routes
const categoriesRouter =require("./routes/categories.route")
app.use('/api/categories', categoriesRouter);
```

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

Tester les services créés :

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

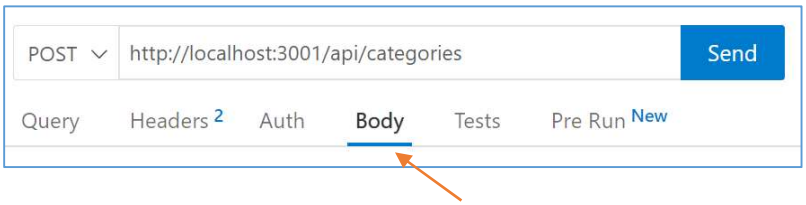
localhost : poste local

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

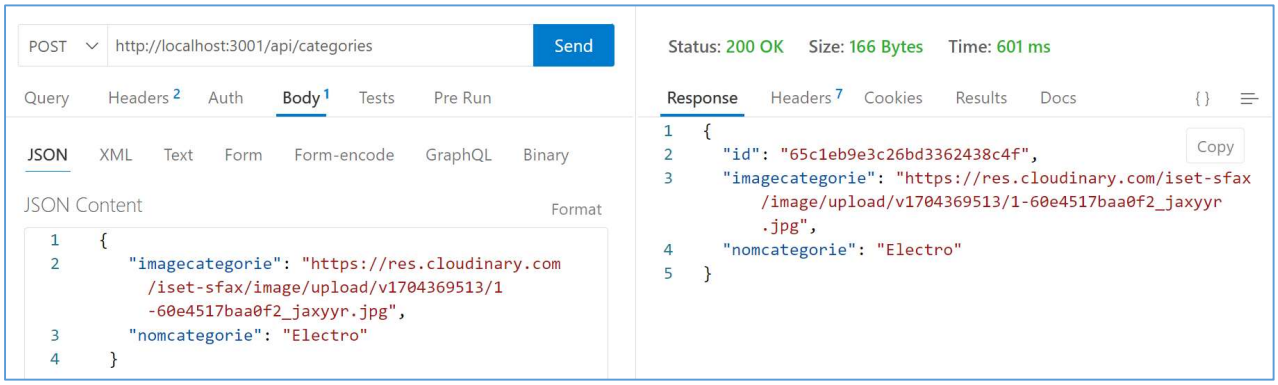
api/categories : la route spécifiée dans app.js : ***app.use('/api/categories', categorieRouter);*** en concaténation avec routes/categorie.route.js ***router.post('/', ...***

La requête pour ajouter est ***POST***

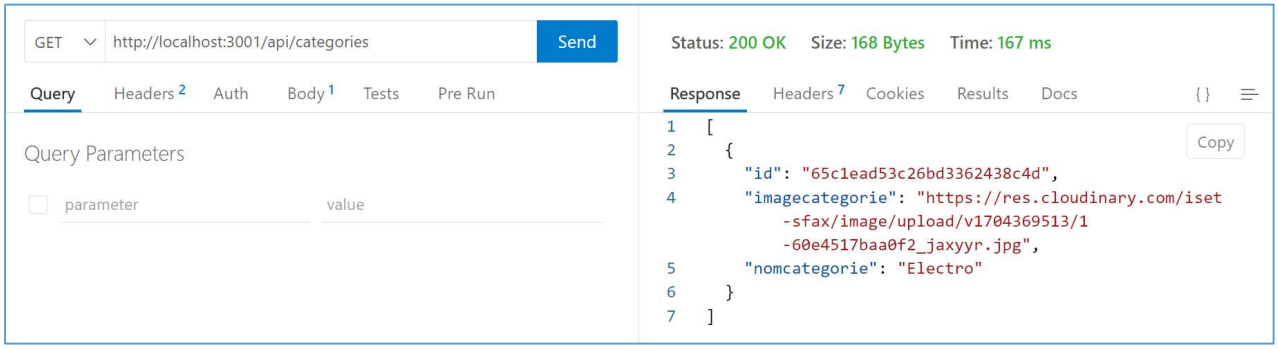
POST http://localhost:3001/api/categories



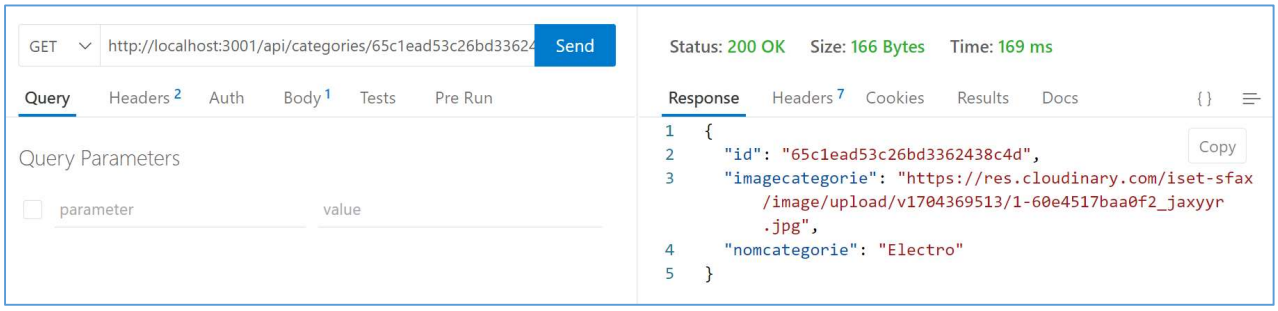
Saisir le code json en respectant le schéma créé.



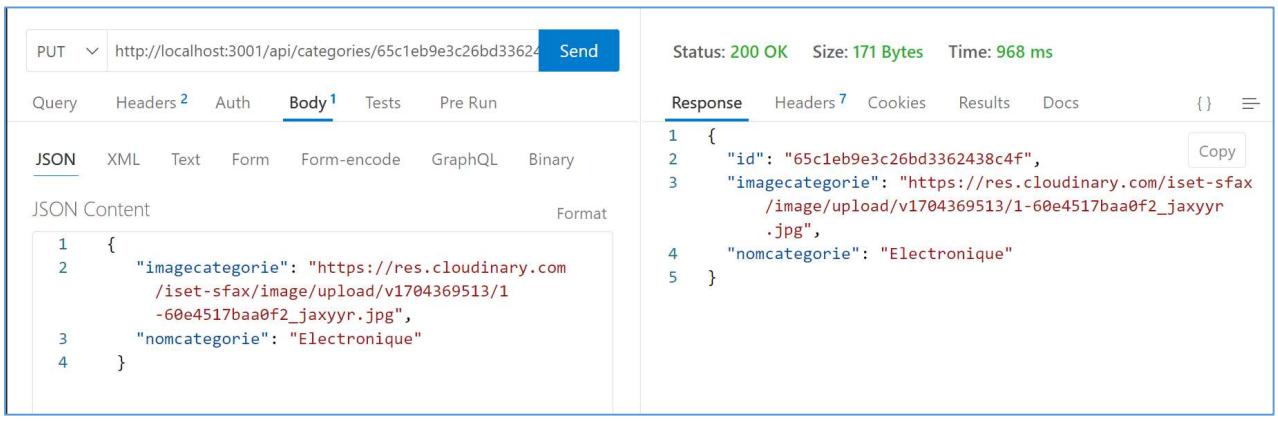
GET http://localhost:3001/api/categories



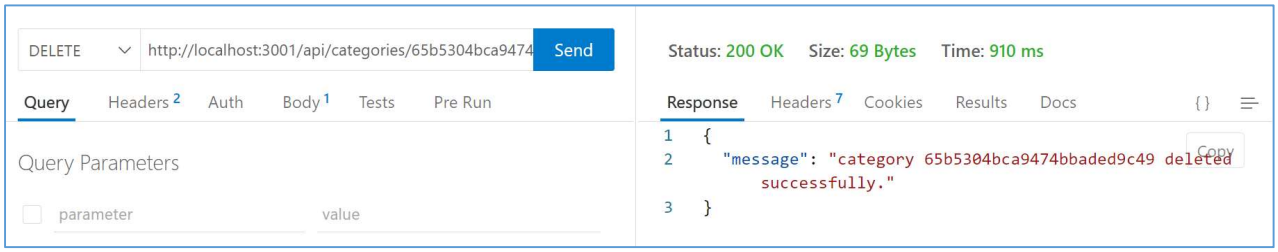
GET http://localhost:3001/api/categories/valeur_id



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



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



II. CRUD Sous categories

Créer le fichier routes/scategories.route.js

routes > JS scategories.route.js >

```
const express = require('express');
const { PrismaClient } = require('@prisma/client')
const prisma = new PrismaClient()
const router = express.Router();

//Ajout d'une scategorie
router.post('/', async (req, res, )=> {
    const {nomscategorie,imagescategorie,categorieID}=req.body
    try {
        const scategorie = await prisma.scategories.create({
            data: {
                nomscategorie: nomscategorie,
                imagescategorie: imagescategorie,
                categorieID:categorieID
            }
        })

        res.json(scategorie)
      } catch (error) {
        res.status(500).json({ message: error.message });
      }

});

// afficher la liste des catégories.
router.get('/', async (req, res, )=> {
    try {
        // scategories est le nom du model précisé dans prisma.schema
        const scategories = await prisma.scategories.findMany({
            include: {
            categories: {
                select: {
                    nomcategorie: true,
                },
```

```
          }
        }
        })
         res.json(scategories)
        } catch (error) {
            res.status(500).json({ message: error.message });
       }
});

// afficher une scategorie.
router.get('/:id', async (req, res, )=> {
    const { id } = req.params

    try {
        const scategorie = await prisma.scategories.findUnique({
            where: {
                id: id,
            }
        })

       res.json(scategorie)
      } catch (error) {
        res.status(500).json({ message: error.message });
      }

});



// modifier une scategorie
router.put('/:id', async (req, res)=> {
    const {nomscategorie,imagescategorie,categorieID}=req.body
    const id  = req.params.id;

    try {
     const scategorie = await prisma.scategories.update({
        data: {
            nomscategorie: nomscategorie,
            imagescategorie: imagescategorie,
            categorieID:categorieID
        },
            where: { id: id},
        })
    res.json(scategorie);
    } catch (error) {
    res.status(404).json({ message: error.message });
    }
});

// Supprimer un scategories
```

```
router.delete('/:id', async (req, res)=> {
    const  id  = req.params.id;
    try {
     await prisma.scategories.delete({
        where: { id: id },
    })

    res.json({ message: "scategory "+ id +" deleted successfully." });
} catch (error) {
    res.status(404).json({ message: error.message });
    }
});

module.exports = router;
```

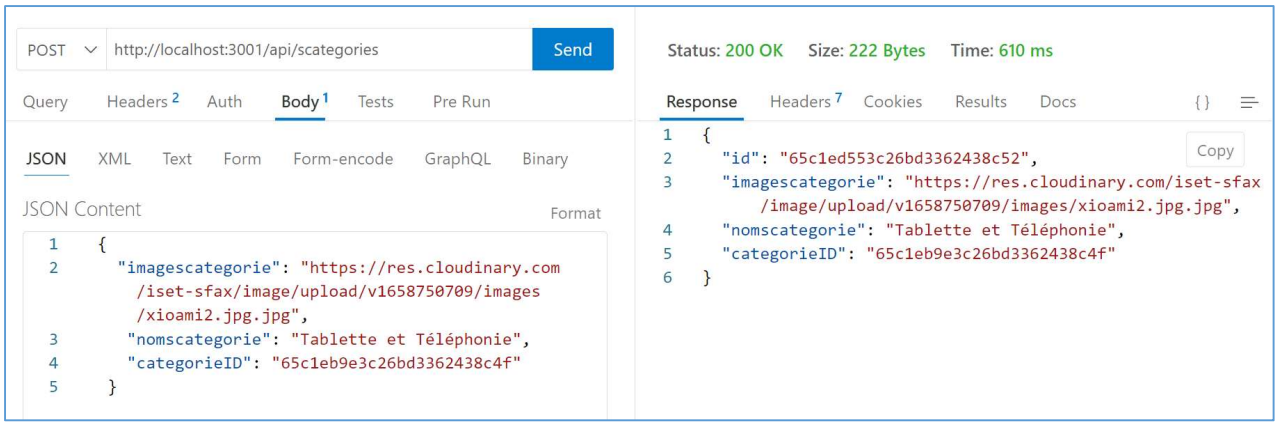
Ajouter la route du fichier dans **app.js**

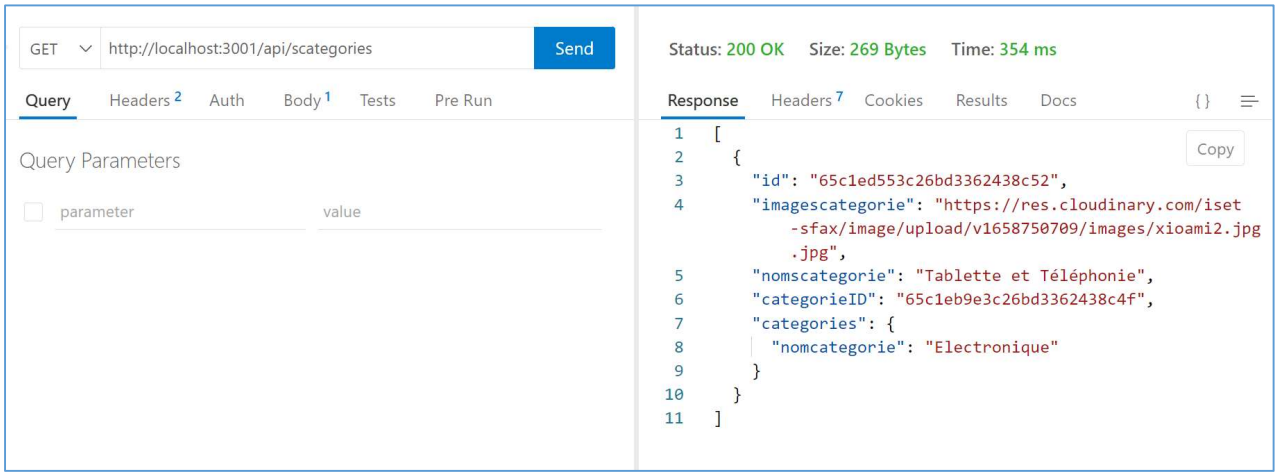
app.js

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

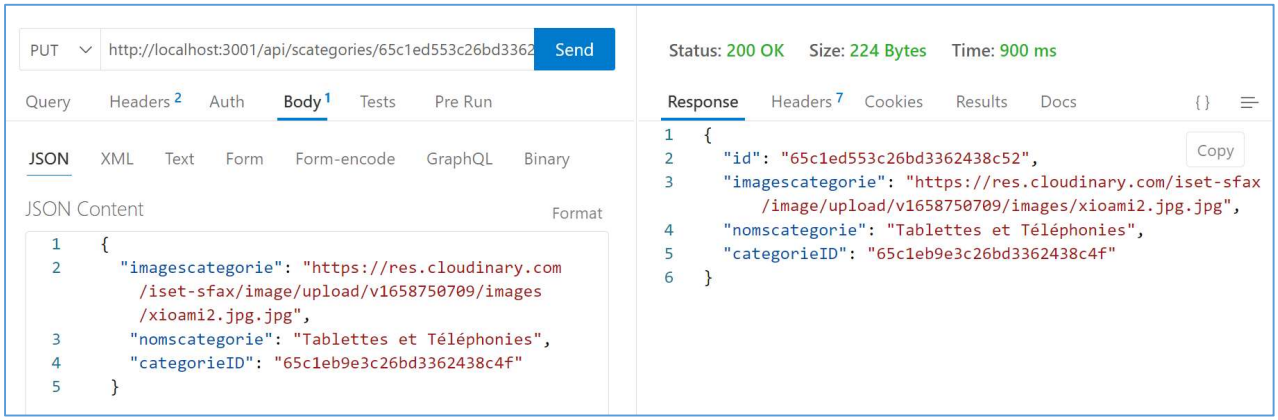
Tester les CRUDs en utilisant **thunder client**

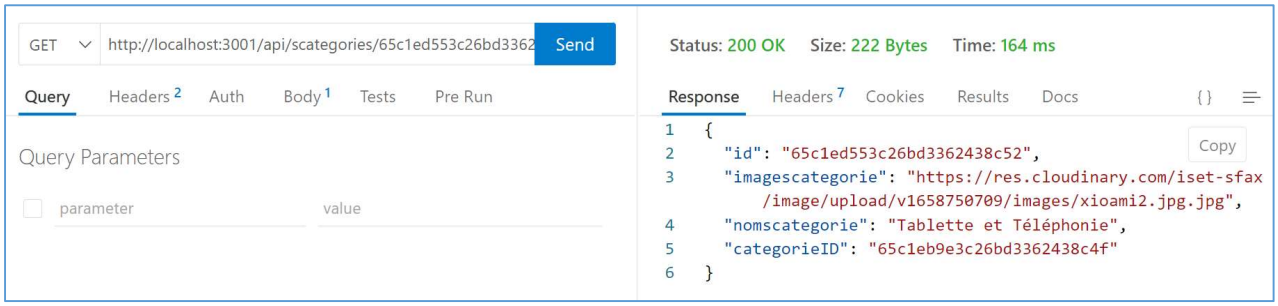
Attention : Tâcher de donner des valeurs existantes dans categories pour categorieID dans la requête POST ou PUT de scategories.

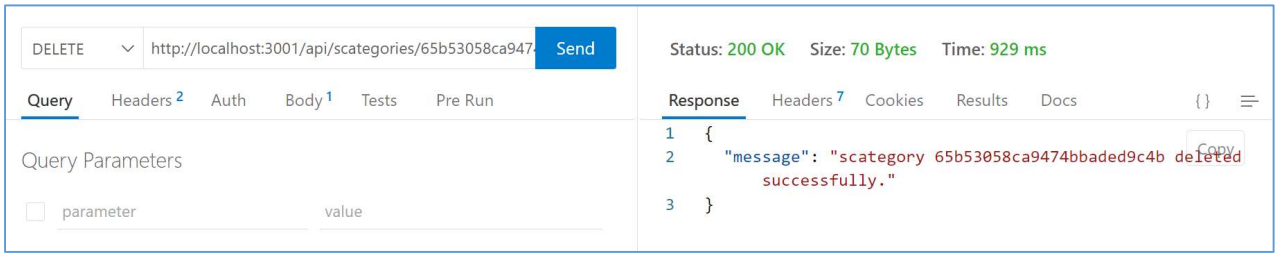




On voit bien le contenu généré de categories grâce à Include dans la requête GET.







III. CRUD articles

Créer le fichier routes/articles.route.js

routes > articles.route.js >

```
const express = require('express');
const { PrismaClient } = require('@prisma/client')
const prisma = new PrismaClient()
const router = express.Router();

//Ajout d'un article
router.post('/', async (req, res, )=> {
    const { designation ,  marque , reference , qtestock , prix , imageart ,
scategorieID}=req.body
    try {
        const article = await prisma.articles.create({
            data: {
                designation : designation,
                marque       : marque,
                reference    : reference,
                qtestock     : qtestock,
                prix         : prix,
                imageart     : imageart,
                scategorieID : scategorieID
            }
        })

        res.json(article)
      } catch (error) {
        res.status(500).json({
          message: "Something went wrong",
          error: error.message,
      });
      }

});

// afficher la liste des articles.
/*
router.get('/', async (req, res, )=> {
    try {
        // articles est le nom du model précisé dans prisma.schema
        const article = await prisma.articles.findMany({
            include: {
                scategories: {
                  include: {
                    categories: true,
                  },
                },
              },
          })
```

```
            res.json(article)

        } catch (error) {
         res.status(500).json({
        message: "Something went wrong",
       })
       }
});
*/
// afficher un article.
router.get('/:id', async (req, res, )=> {
    const { id } = req.params

    try {
        const article = await prisma.articles.findUnique({
            where: {
                id: id,
            }
        })

       res.json(article)
      } catch (error) {
        res.status(500).json({
          message: "Something went wrong",
        })
      }

});



// modifier un article
router.put('/:id', async (req, res)=> {
    const { designation ,  marque , reference , qtestock , prix , imageart ,
scategorieID }=req.body
    const id  = req.params.id;

    try {
     const article = await prisma.articles.update({
        data: {
            designation : designation,
            marque       : marque,
            reference    : reference,
            qtestock    : qtestock,
            prix        : prix,
            imageart     : imageart,
            scategorieID : scategorieID
        },
            where: { id: id},
        })
```

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

// Supprimer un article
router.delete('/:id', async (req, res)=> {
    const  id  = req.params.id;
    try {
     await prisma.articles.delete({
        where: { id: id },
    })

    res.json({ message: "article "+ id +" deleted successfully." });
} catch (error) {
    res.status(404).json({ message: error.message });
    }
});


router.get('/', async (req, res) => {

   try {
        const page_str = req.query.page;
    const limit_str = req.query.limit;

    const page = page_str ? parseInt(page_str, 10) : 1;
    const limit = limit_str ? parseInt(limit_str, 10) : 10;
    const skip = (page - 1) * limit;
      const articles = await prisma.articles.findMany({
        skip,
        take: limit,
        include: {
          scategories: {
            include: {
              categories: true,
            },
          },
        },
      });

      res.json(articles);
    } catch (error) {
      res.status(500).json({
        message: "Something went wrong",
        error: error.message,
      });
    }
  });
```

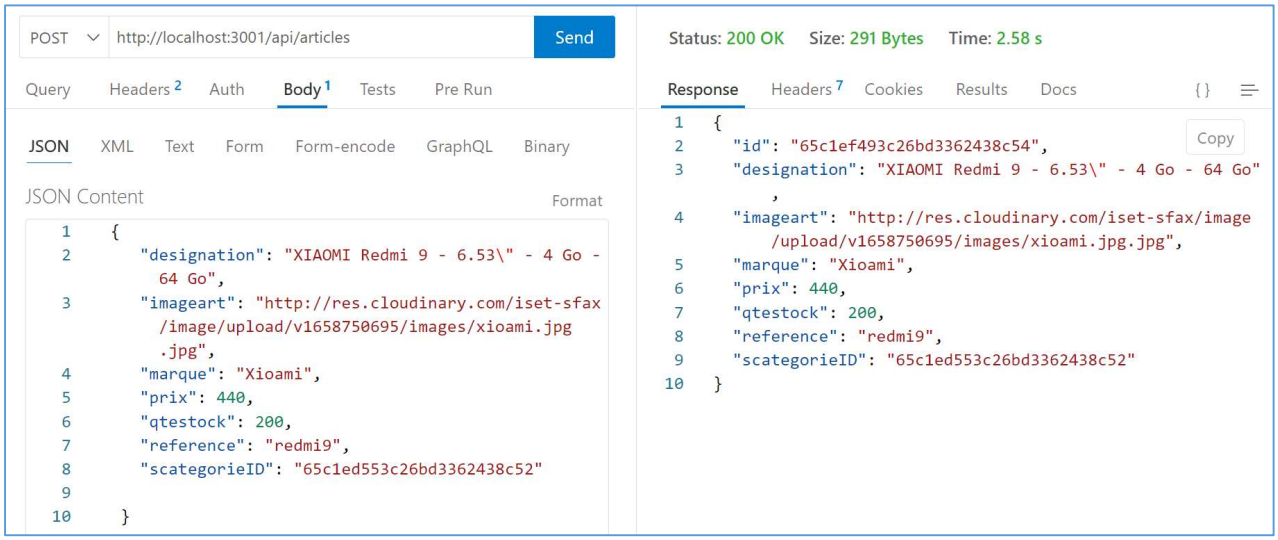
```
  //liste des articles pour une catégorie donnée
  router.get('/cat/:idCateg', async (req, res) => {
    const { idCateg } = req.params;

    try {
        const articles = await prisma.articles.findMany({
            where: {
                scategories: {
                    categorieID: idCateg
                }
            },
            include: {
                scategories: true,
              },
        });

        res.json(articles);
    } catch (error) {
        res.status(500).json({
            message: "Something went wrong",
            error: error.message,
        });
    }
});

module.exports = router;
```

La requête GET en commentaire est simple sans pagination ni limite de nombre d'enregistrement. Elle a été remplacée par la requête utilisant limit et skip. Si on ne les précise pas dans la requête on prendra par défaut la page 1 et 10 enregistrements comme limite.

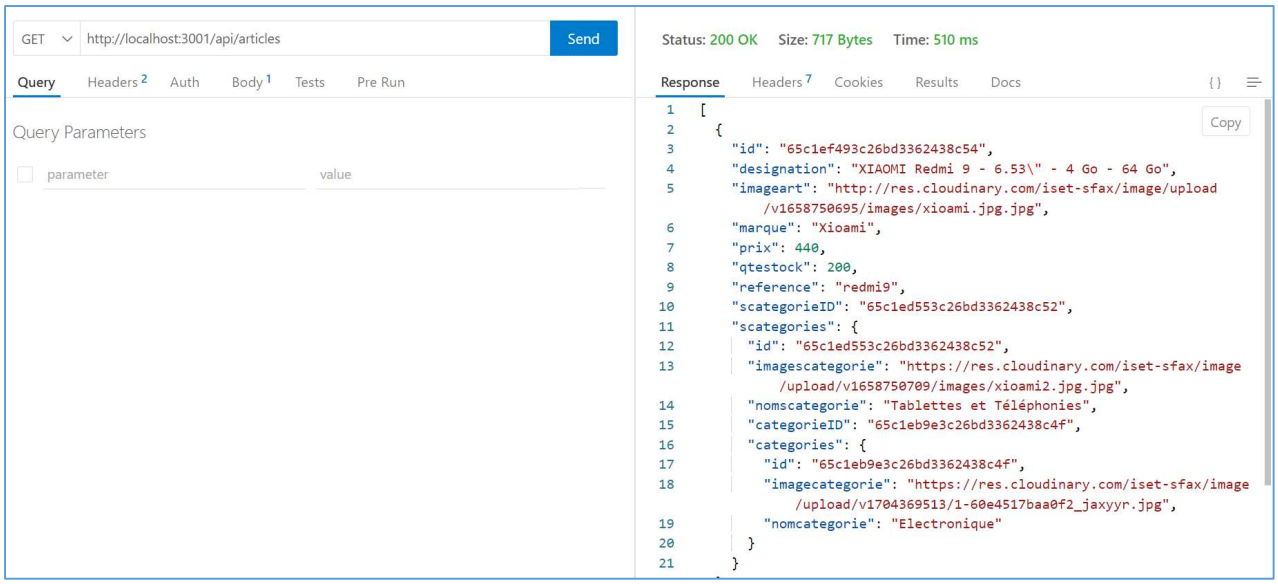
Dans le fichier **app.js** ajouter la route de l'article.


```
const articlesRouter =require("./routes/articles.route")
app.use('/api/articles', articlesRouter);
```

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

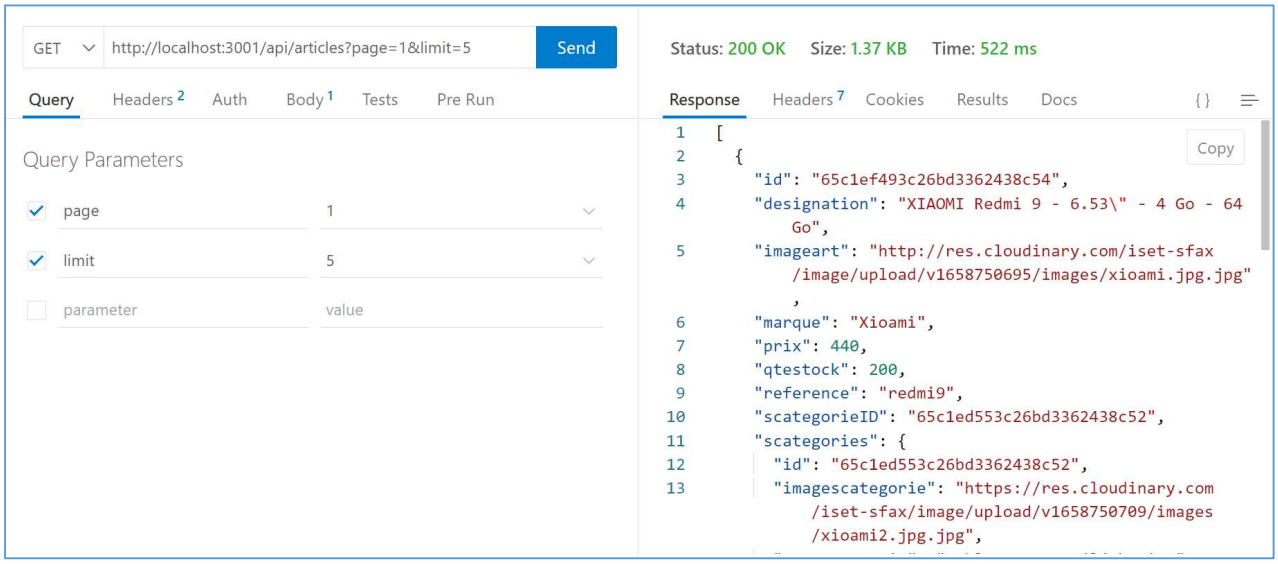


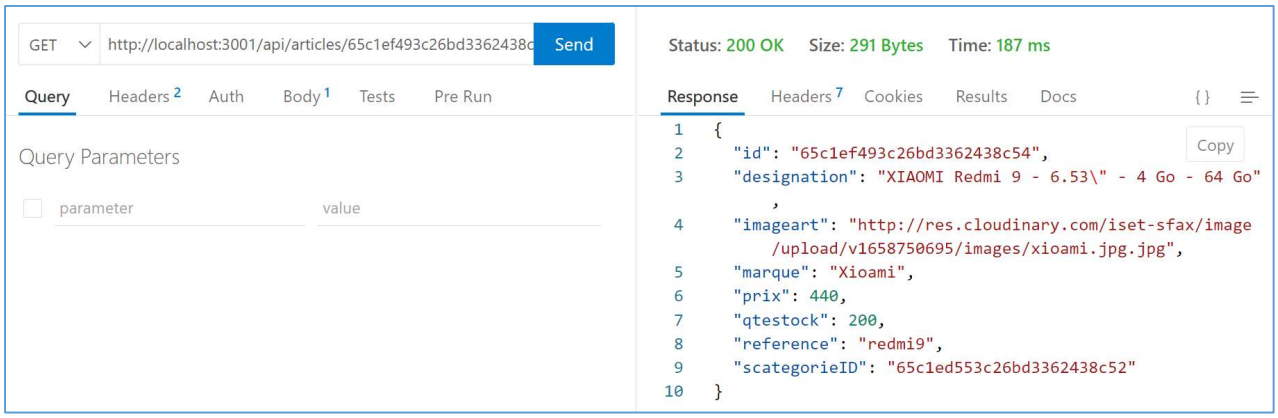
Exemple de résultat où on voit bien le contenu généré de scategories grâce à Include dans la requête GET.

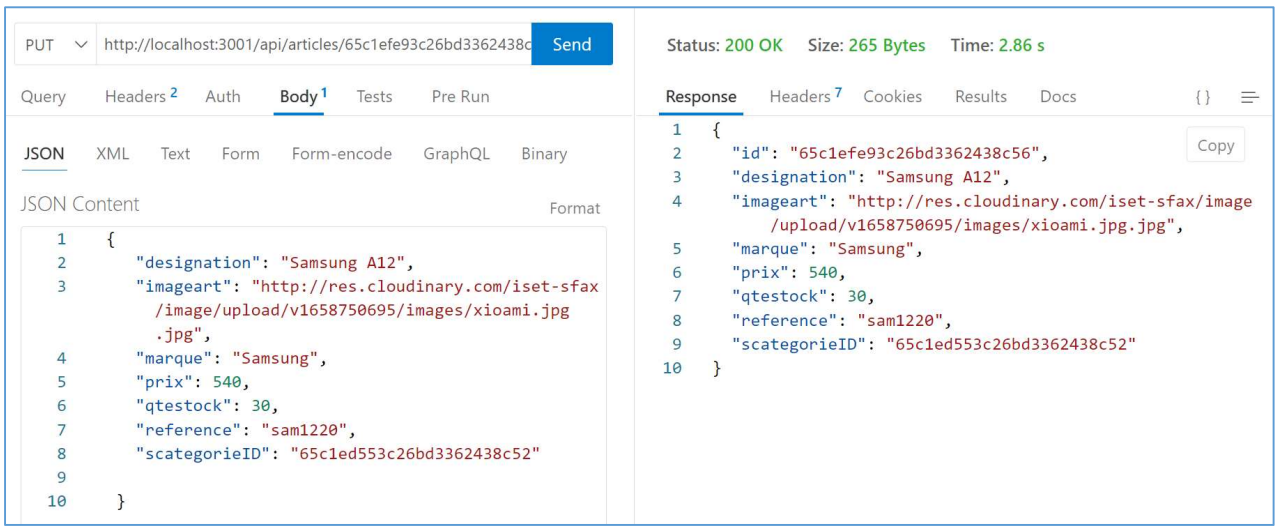


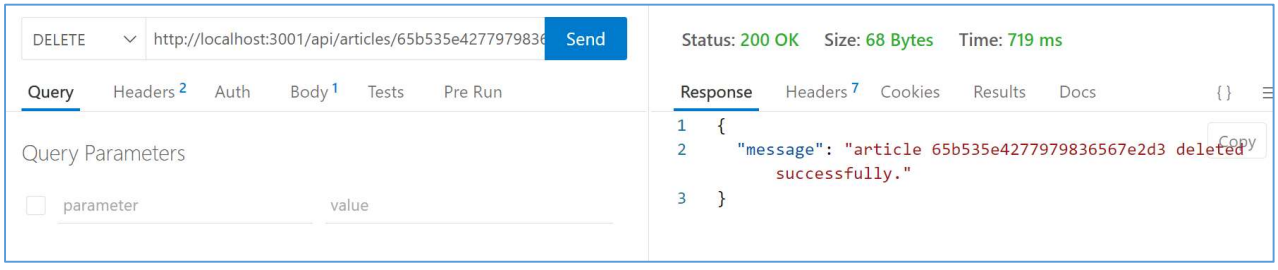
On va préciser la page et la limite dans la requête GET

http://localhost:3001/api/articles?page=1&limit=5


GET
http://localhost:3001/api/articles?page=1&limit=5
Send
Status: 200 OK
Size: 1.37 KB
Time: 522 ms
Query
Headers 2
Auth
Body 1
Tests
Pre Run
Response
Headers 7
Cookies
Results
Docs
Query Parameters
page
1
limit
5
parameter
value
Copy
[
{
"id": "65c1ef493c26bd3362438c54",
"designation": "XIAOMI Redmi 9 - 6.53\" - 4 Go - 64 Go",
"imageart": "http://res.cloudinary.com/iset-sfax/image/upload/v1658750695/images/xioami.jpg.jpg",
"marque": "Xioami",
"prix": 440,
"qtestock": 200,
"reference": "redmi9",
"scategorieID": "65c1ed553c26bd3362438c52",
"scategories": {
"id": "65c1ed553c26bd3362438c52",
"imagescategorie": "https://res.cloudinary.com/iset-sfax/image/upload/v1658750709/images/xioami2.jpg.jpg",


GET
http://localhost:3001/api/articles/65c1ef493c26bd3362438c
Send
Status: 200 OK
Size: 291 Bytes
Time: 187 ms
Query
Headers 2
Auth
Body 1
Tests
Pre Run
Response
Headers 7
Cookies
Results
Docs
Query Parameters
parameter
value
Copy
{
"id": "65c1ef493c26bd3362438c54",
"designation": "XIAOMI Redmi 9 - 6.53\" - 4 Go - 64 Go",
"imageart": "http://res.cloudinary.com/iset-sfax/image/upload/v1658750695/images/xioami.jpg.jpg",
"marque": "Xioami",
"prix": 440,
"qtestock": 200,
"reference": "redmi9",
"scategorieID": "65c1ed553c26bd3362438c52"
}


PUT
http://localhost:3001/api/articles/65c1efe93c26bd3362438c
Send
Status: 200 OK
Size: 265 Bytes
Time: 2.86 s
Query
Headers 2
Auth
Body 1
Tests
Pre Run
Response
Headers 7
Cookies
Results
Docs
JSON
XML
Text
Form
Form-encode
GraphQL
Binary
JSON Content
Format
Copy
{
"designation": "Samsung A12",
"imageart": "http://res.cloudinary.com/iset-sfax/image/upload/v1658750695/images/xioami.jpg.jpg",
"marque": "Samsung",
"prix": 540,
"qtestock": 30,
"reference": "sam1220",
"scategorieID": "65c1ed553c26bd3362438c52"
}


DELETE
http://localhost:3001/api/articles/65b535e4277979836
Send
Status: 200 OK
Size: 68 Bytes
Time: 719 ms
Query
Headers 2
Auth
Body 1
Tests
Pre Run
Response
Headers 7
Cookies
Results
Docs
Query Parameters
parameter
value
Copy
{
"message": "article 65b535e4277979836567e2d3 deleted successfully."
}