

Cours : **Base de Donnée NoSQL**

THEME : Réalisation d'une API RESTful connectée à
MongoDB Atlas

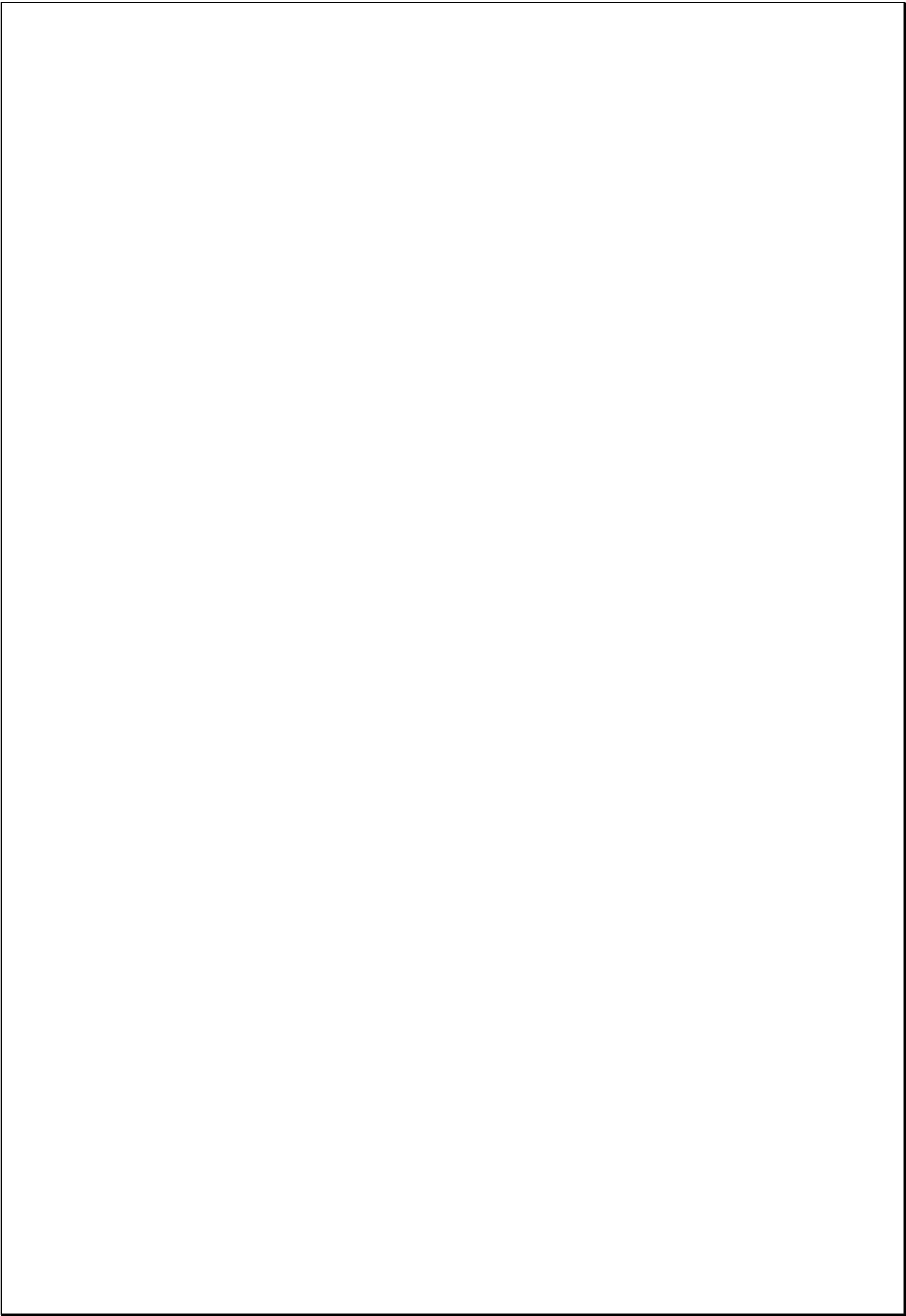
Membres :

DIALLO Ousmane Sidi M.S B-W-S

Professeur :

Mr Jean-Marie PREIRA

Année Académique 2021-2022



PLAN

Liste des figures.....	4
1- INTRODUCTION	1
1- Qu'est-ce qu'une API ?.....	1
2.1- Différents types d'API	1
2.2- REST.....	2
2- Présentation de l'implémentation du projet	3
3.1- Choix et description du jeu de donnée.....	3
3.2- Liste des requêtes	4
3.3- Description de la procédure de déploiement de la base de données sur Mongo Atlas	5
3.4- Présentation et explication des codes sources écrits.....	11
3.5- Documentation de l'API	18
3- Conclusion.....	19

Liste des figures

Figure 1: Création de projet sur MongoDB Atlas	5
Figure 2: Attribution du nom du projet	5
Figure 3: Génération du projet	6
Figure 4: Construction de la base de données	6
Figure 5: Différentes Options d'hébergement	7
Figure 6: Choix du fournisseur de service cloud.....	7
Figure 7: Remplissage des informations d'identification	8
Figure 8: Choix de l'environnement d'accès aux données	8
Figure 9: Informations du cluster	9
Figure 10: Connect to Cluster	10
Figure 11: Connexion à MongoDB Compass	10
Figure 12: Schéma de la base de données	11
Figure 13: Requête localhost :3000/api/post.....	12
Figure 14: Requête localhost :3000/api/getALL.....	13
Figure 15: Requête localhost :3000/api/getOne/:id.....	13
Figure 16: Requête localhost:3000/api/getBySymbole/:Symbol	14
Figure 17: Requête localhost:3000/api/getBySymbolVolumeTotal/:Symbol.....	14
Figure 18: Requête localhost:3000/api/getByGroupSymbol	15
Figure 19: Requête localhost:3000/api/update/:id.....	15
Figure 20: Requête localhost:3000/api/updateBySymbol/:Symbol	16
Figure 21: Requête localhost:3000/api/delete/:id.....	16
Figure 22: Requête localhost:3000/api/deleteBySymbol/:Symbol	17
Figure 23: Importation de module.....	17
Figure 24: Connexion à la Base de Donnée	18

1- INTRODUCTION

Les API deviennent de plus en plus incontournables au vu de leur employabilité dans le secteur informatique. Ils permettent d'avoir accès à des informations très importantes, d'utiliser des fonctionnalités sans avoir à les développer ou à refaire tout le chemin de collecte d'information. Nous allons définir et expliquer comment fonctionne un API et détailler son implémentation.

1- Qu'est-ce qu'une API ?

Une API (Application Programming Interface) est un programme permettant à deux applications distinctes de **communiquer entre elles** et **d'échanger des données**. Cela évite notamment de recréer et redévelopper entièrement une application pour y ajouter ses informations. Par exemple, elle est là pour **faire le lien** entre des **données déjà existantes** et un **programme indépendant**. Supposons que l'on a pour objectif de développer une application météo, ce n'est pas l'application qui va chercher et analyser les informations météorologiques et vous les transmettre mais bien une API qui va se connecter à la base de données où se trouvent ces informations pour les afficher dans votre application. Un autre exemple est que, si l'on souhaite intégrer une carte Google sur un site Internet, on utilisera une API Google Maps mise à disposition par le géant américain GOOGLE.

2.1- Différents types d'API

Il existe **deux catégories principales d'API** à savoir les **API publiques**, appelées aussi open API, et les **API privées**, connues sous le nom de entreprise API. Les API sécurisées disposent alors d'une **clé d'identification**, fournie par un service d'authentification et d'autorisation.

Du fait de la grande diversité d'applications client, les API doivent s'appuyer sur un protocole de communication, le **SOAP** (Simple Object Access Protocol) ou le **REST** (Representational State Transfert) afin d'être **compatibles aux diverses**

plateformes mobiles, qu'il s'agisse d'une application Windows, Apple ou Android. L'API Rest (ou Restful) est à présent la plus utilisée car elle offre plus de flexibilité. Dans le cadre de notre projet c'est le type d'API qui nous implémenterons.

2.2- REST

Une API REST (également appelée API RESTful) est une interface de programmation d'application (API ou API web) qui respecte les contraintes du style d'architecture REST et permet d'interagir avec les services web RESTful. REST est un ensemble de contraintes architecturales. Il ne s'agit ni d'un protocole, ni d'une norme.

Une API RESTful doit remplir les critères suivants :

- Une architecture client-serveur constituée de clients, de serveurs et de ressources, avec des requêtes gérées via HTTP
- Des communications client-serveur **stateless**, c'est-à-dire que les informations du client ne sont jamais stockées entre les requêtes GET, qui doivent être traitées séparément, de manière totalement indépendante
- La possibilité de mettre en cache des données afin de rationaliser les interactions client-serveur
- Une interface uniforme entre les composants qui permet un transfert standardisé des informations Cela implique que :
 - Les ressources demandées soient identifiables et séparées des représentations envoyées au client ;
 - Les ressources puissent être manipulées par le client au moyen de la représentation reçue, qui contient suffisamment d'informations ;
 - Les messages autodescriptifs renvoyés au client contiennent assez de détails pour décrire la manière dont celui-ci doit traiter les informations ;
 - L'API possède un hypertexte/hypermédia, qui permet au client d'utiliser des hyperliens pour connaître toutes les autres actions disponibles après avoir accédé à une ressource.
- Un système à couches, invisible pour le client, qui permet de hiérarchiser les différents types de serveurs (pour la sécurité, l'équilibrage de charge, etc.) impliqués dans la récupération des informations demandées

- Du code à la demande (facultatif), c'est-à-dire la possibilité d'envoyer du code exécutable depuis le serveur vers le client (lorsqu'il le demande) afin d'étendre les fonctionnalités d'un client

Puisque REST est un ensemble de directives mises en œuvre à la demande, les API REST sont plus rapides et légères, et offrent une évolutivité accrue.

Dans le cadre de notre projet c'est le type d'API qui nous implémenterons.

2- Présentation de l'implémentation du projet

3.1- Choix et description du jeu de donnée

Notre jeu de donnée contient des informations sur le prix de stock concernant le top 4 des compagnies françaises de luxe entre 2000-2022 à savoir :

RMS.PA = Hermes

MC.PA = Louis Vuitton

CDL.PA = Christian Dior

KER.PA = Kering

Les informations que reflètent les différentes colonnes sont :

Date : date de transaction

Symbol : symbole de la compagnie

Adj Close : Adj Close pour Adjusted Close est le cours de fermeture ajustée. Il modifie le cours de clôture d'une action après avoir pris en compte toutes les opérations sur les titres.

Close : Représente le dernier prix négocié avant la fermeture du marché

High : prix maximum d'une action sur une certaine période de temps.

Low : prix minimum d'une action sur une certaine période de temps.

Open : Représente le premier prix négocié lors de l'ouverture du marché

Volume : est le montant d'un actif ou d'un titre qui change de main sur une certaine période de temps, généralement la journée. Par exemple, le volume des transactions boursières fait référence au nombre d'actions d'un titre négociées entre son ouverture et sa clôture quotidienne.

3.2- Liste des requêtes

Nous avons utilisé un total de 09 requêtes dont :

- ❑ 01 requête d'écriture POST :

```
curl --location --request POST 'localhost:3000/api/post'
```

- ❑ 02 requêtes de suppression DELETE :

```
✓ curl --location --request DELETE  
  'localhost:3000/api/delete/628d83ecc9b5b2c4e1bc574c'  
✓ curl --location --request DELETE  
  'localhost:3000/api//deleteBySymbol/CDI.PA'
```

- ❑ 02 requêtes de mise à jour (modification) PATCH :

```
✓ curl --location --request PATCH  
  'localhost:3000/api/update/628d84d8f72ff884722ba3f9'  
✓ curl --location --request PATCH  
  'localhost:3000/api/updateBySymbol/CDI.PA'
```

- ❑ 04 requêtes de lecture GET :

```
✓ curl --location --request GET  
  'localhost:3000/api/getOne/628d84d8f72ff884722ba3e0'  
✓ curl --location --request GET  
  'localhost:3000/api/getBySymbole/CDI.PA'  
✓ curl --location --request GET  
  'localhost:3000/api/getByGroupSymbol'  
✓ curl --location --request GET  
  'localhost:3000/api/getBySymbolVolumeTotal/ESMT'
```


3.3- Description de la procédure de déploiement de la base de données sur Mongo Atlas

Dans cette partie, nous allons énumérer les étapes à suivre pour déployer la base de données sur Mongo Atlas.

- Etape 1 : Connexion à MongoDB Atlas. Créer un compte au préalable

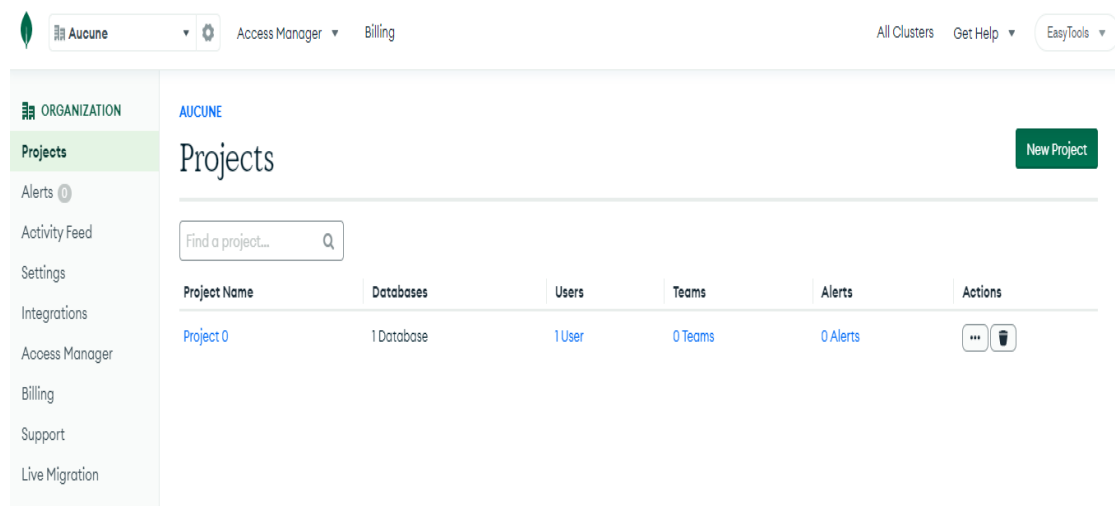


Figure 1: Création de projet sur MongoDB Atlas

- Etape 2 : Créer un projet. Aller dans l'onglet **Projets**. Dans cette partie, on nomme notre projet.

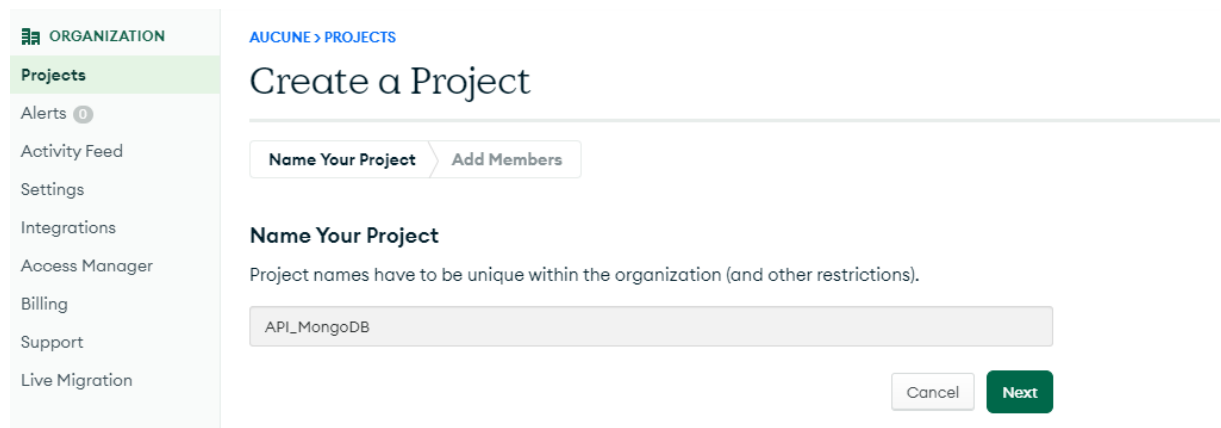


Figure 2: Attribution du nom du projet

- Etape 3 : Dans cette partie, on peut soit ajouter des membres et fixer les permissions ou directement créer notre **projet**.

The screenshot shows the 'Create a Project' page in the MongoDB Atlas console. On the left is a sidebar with the 'ORGANIZATION' menu, where 'Projects' is selected. The main area has a header 'Create a Project' and two tabs: 'Name Your Project' (active) and 'Add Members'. Below the tabs is a section 'Add Members and Set Permissions' with an input field for email addresses. A member 'diallofulanisidi@gmail.com (you)' is listed with the role 'Project Owner'. At the bottom are buttons for 'Cancel', 'Go Back', and 'Create Project'.

Figure 3: Génération du projet

- Etape 4 : Nous allons construire la base de donnée

The screenshot shows the 'Database Deployments' page in the MongoDB Atlas console. The left sidebar has the 'DEPLOYMENT' menu, with 'Database' selected. The main area has a header 'Database Deployments' and a large green button 'Build a Database'. Below the button is a note: 'Once your database is up and running, live migrate an existing MongoDB database into Atlas with our [Live Migration Service](#).' The breadcrumb 'AUCUNE > API_MONGODB' is visible at the top.

Figure 4: Construction de la base de données

- Etape 5 : Trois options sont possibles dans cette partie. L'option qui nous concerne est l'option gratuite qui nous permettra de créer un cluster.

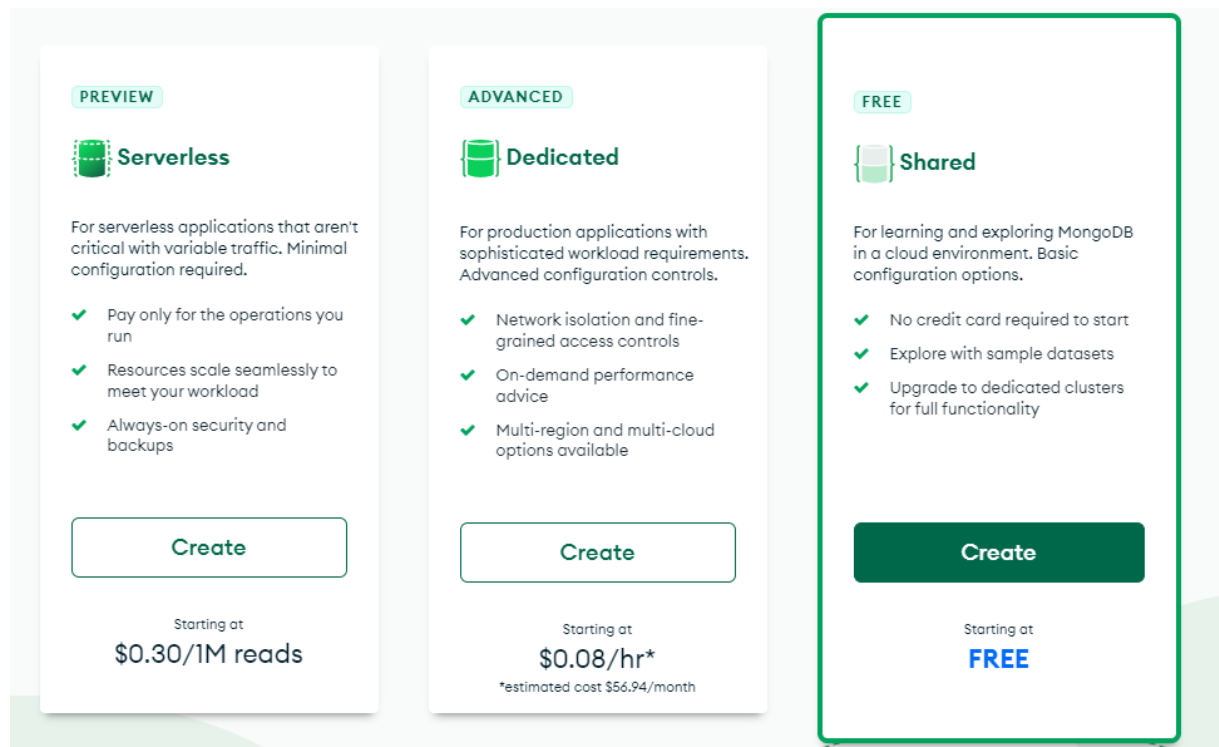


Figure 5: Différentes Options d'hébergement

- Etape 6 : Nous choisissons la zone d'hébergement et le fournisseur du service cloud

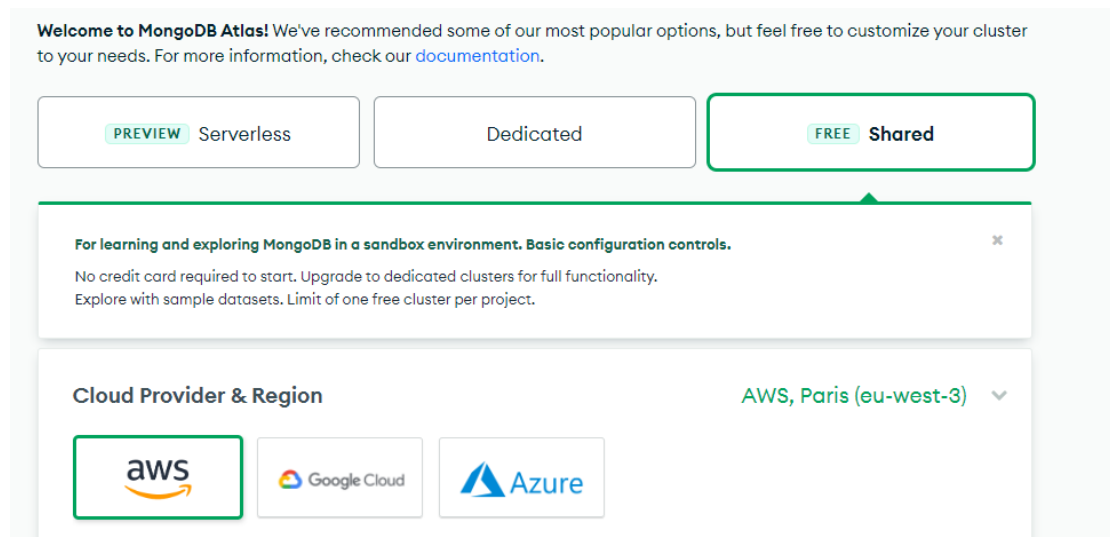
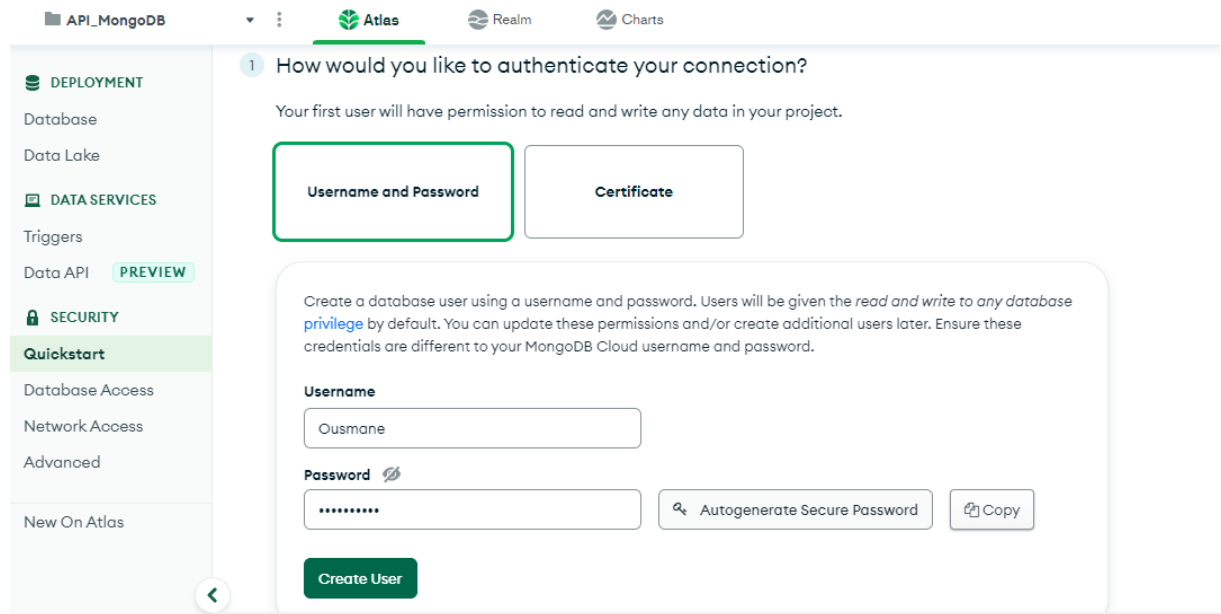


Figure 6: Choix du fournisseur de service cloud

- Etape 7 : Le premier utilisateur que nous allons créer a les droits d'écriture et de lecture sur la base de donnée. L'option **Autogenerate secure password** permet de générer un mot de passe.



API_MongoDB Atlas Realm Charts

DEPLOYMENT

Database

Data Lake

DATA SERVICES

Triggers

Data API PREVIEW

SECURITY

Quickstart

Database Access

Network Access

Advanced

New On Atlas

1 How would you like to authenticate your connection?

Your first user will have permission to read and write any data in your project.

Username and Password Certificate

Create a database user using a username and password. Users will be given the *read and write to any database privilege* by default. You can update these permissions and/or create additional users later. Ensure these credentials are different to your MongoDB Cloud username and password.

Username

Ousmane

Password

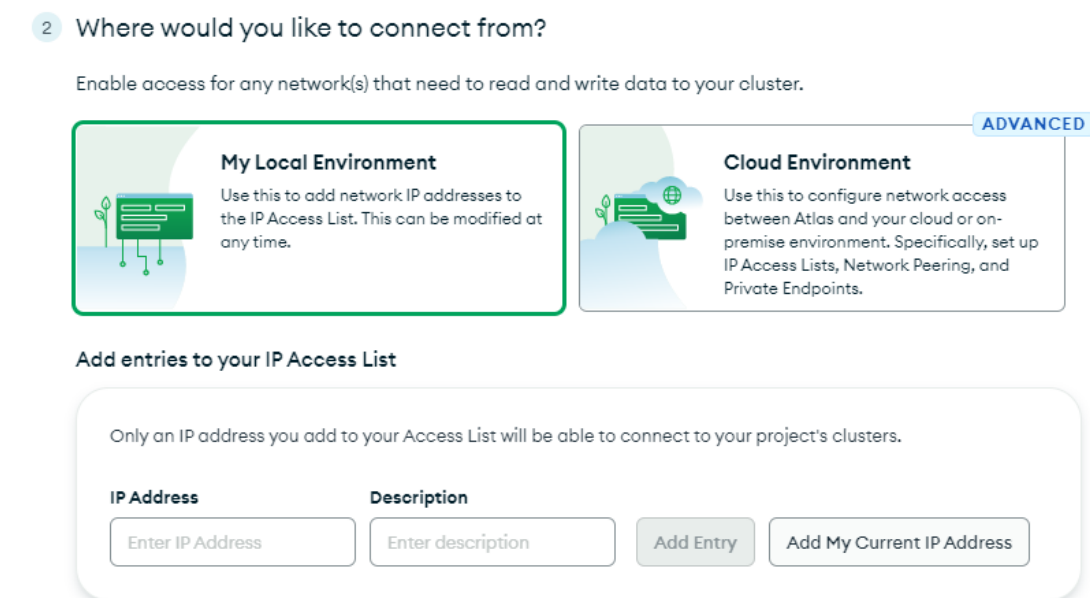
Autogenerate Secure Password

Copy

Create User

Figure 7: Remplissage des informations d'identification

- Etape 8 : Nous pouvons choisir de nous connecter en environnement local ou par environnement cloud. Notre choix s'est porté sur l'environnement local
- Nous pouvons fixer ou générer notre adresse ip de connexion.



2 Where would you like to connect from?

Enable access for any network(s) that need to read and write data to your cluster.

My Local Environment

Use this to add network IP addresses to the IP Access List. This can be modified at any time.

Cloud Environment

Use this to configure network access between Atlas and your cloud or on-premise environment. Specifically, set up IP Access Lists, Network Peering, and Private Endpoints.

ADVANCED

Add entries to your IP Access List

Only an IP address you add to your Access List will be able to connect to your project's clusters.

IP Address	Description
Enter IP Address	Enter description

Add Entry

Add My Current IP Address

Figure 8: Choix de l'environnement d'accès aux données

- Etape 9 : On clique sur le bouton **Connect**

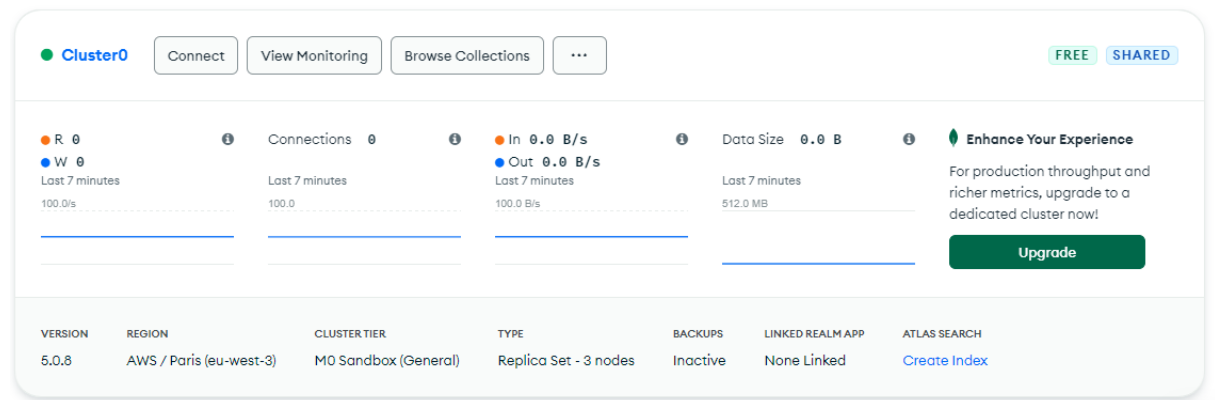
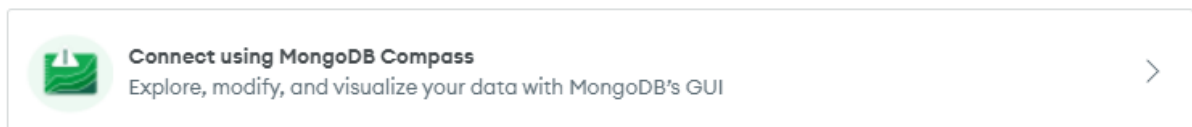


Figure 9: Informations du cluster

- Etape 10 : Dans la nouvelle fenêtre qui apparaît, nous choisissons la connexion par MongoDB Compass



- Etape 11 : Etant donné que MongoDB Compass est déjà installé, nous avons choisi la deuxième option. Ensuite, on copie l'URI de connexion (connection string).

Connect to Cluster0

Figure 10: Connect to Cluster

- Etape 12 : On ouvre l'application MongoDB Compass et on colle le lien de connexion. On remplace <password> par le mot de passe (Propre à l'utilisateur)

Figure 11: Connexion à MongoDB Compass

Après s'être connecté à MongoDB Compass, nous pouvons créer des bases de données.

3.4- Présentation et explication des codes sources écrits

Notre projet a été développé dans un environnement javascript (nodejs). Notre code source est subdivisé en trois fichiers javascript et un fichier env qui sont :

- model.js
- routes.js
- index.js
- .env

- **Fichier model.js** : contient le schema de la base de données

```
1  const mongoose = require('mongoose');
2  const databaseSchema = new mongoose.Schema({
3    >   Date: { ...
6    >   },
7    >   Symbol: { ...
10   >   },
11  >   "Adj Close": { ...
14   >   },
15  >   Close: { ...
18   >   },
19  >   High: { ...
22   >   },
23  >   Low: { ...
26   >   },
27  >   Open: { ...
30   >   },
31  >   Volume: { ...
34   >   }
35  > })
36
37  module.exports = mongoose.model('Data', databaseSchema);
```

Figure 12: Schéma de la base de données

Nous importons la totalité du module **mongoose** à l'aide de la commande **require**.

Ensuite nous définissons une constante qui récupère le schema de la base de données.

Les champs Date et Symbol sont de type **String** et sont **indispensables**, donc on indique ces informations par les commandes suivantes.

```
4      required: true,  
5      type: String
```

Les champs Adj Close, Close, High, Low, Open et Volume sont de type **Number** et sont **indispensable**, donc on indique ces informations par les commandes suivantes.

```
16     required: true,  
17     type: Number
```

- **Fichier routes.js** : contient les différentes requêtes offertes par l'API
 - ✓ **Requête 1** : cette requête va nous permettre d'insérer un document. Nous définissons une variable **data** qui est un objet de type **Model** par la commande **new** et la classe **Model**.

```
12 //Ajouter un document  
13 route.post('/post', async (req,res)=>{  
14     const data = new Model({  
15         Date: "2022-04-29",  
16         Symbol: "ESMT",  
17         "Adj Close": 34,  
18         Close:35,  
19         High:50,  
20         Low: 37.456,  
21         Open: 45,  
22         Volume: 48.555  
23     });  
24  
25     try{  
26         const dataTosave = await data.save();  
27         res.status(200).json(dataTosave)  
28     }catch(error){  
29         res.status(400).json({message:error.message})  
30     }  
31 });
```

Figure 13: Requête localhost :3000/api/post

Nous utilisons la fonction **save(argument)** pour insérer le document. La fonction **status()** de réponse **res** prends comme attribut la réponse http adéquate.

200 : indique que la réponse a été accomplie correctement

400 : indique que la syntaxe de la requête est mal formulée ou est impossible à satisfaire.

La fonctions **json(argument)** convertit l'argument en document json.

- ✓ **Requête 2** : la fonction **find()** permet de renvoyer tous les documents de la base de donnée.

```
33 //Obtenir tous les documents de la collection
34 route.get('/getAll', async (req,res)=>{
35     try{
36         const data = await Model.find();
37         res.json(data);
38     }catch(error){
39         res.json({message:error.message});
40     }
41 });
```

Figure 14: Requête localhost :3000/api/getALL

- ✓ **Requête 3** : Nous utilisons la fonction **findById(id)**, qui retourne le ou les documents qui ont le même id indiquer en argument. On récupère l'id par commande **req.params.id**

```
43 //Obtenir un document par son id
44 route.get('/getOne/:id', async (req,res)=>{
45     try{
46         const dataById = await Model.findById(req.params.id);
47         res.json(dataById);
48     }catch(error){
49         res.json({message:error.message});
50     }
51 });
```

Figure 15: Requête localhost :3000/api/getOne/:id

- ✓ **Requête 4** : Nous utilisons la fonction **find()**, qui retourne le ou les documents qui respectent les informations passées en argument. L'argument est {Symbol :value}. Donc tous les documents qui ont pour symbol, **value** seront affiché. On récupère le Symbol par commande **req.params.Symbol**

```
53 //Obtenir un(ou les) document(s) concerné par le symbole
54 route.get('/getBySymbole/:Symbol', async (req,res)=>{
55     try{
56         const value = req.params.Symbol
57         console.log(value)
58         const dataById = await Model.find({Symbol:value});
59         res.json(dataById);
60     }catch(error){
61         res.json({message:error.message});
62     }
63 });
```

Figure 16: Requête localhost:3000/api/getBySymbole/:Symbol

- ✓ **Requête 5** : Pour effectuer le regroupement, nous allons établir le **pipeline** qui est la variable **aggregation**. Dans ce pipeline, nous allons filtrer des documents concernés grâce à **\$match**, puis regrouper en sommant les volumes appartenant à un même symbole grace à **\$group** et **\$sum**. Puis, nous utilisons la fonction **aggregate()** qui prendra en argument notre pipeline.

```
65 //Afficher le volume total par symbole choisi
66 route.get('/getBySymbolVolumeTotal/:Symbol', async (req,res)=>{
67     try{
68         const value = req.params.Symbol
69         const aggregation = [
70             {$match : {Symbol:value}},
71             {$group: {_id:value, totalVolume:{$sum:"$Volume"}}}
72         ]
73         const dataById = await Model.aggregate(aggregation);
74         res.json(dataById);
75     }catch(error){
76         res.json({message:error.message});
77     }
78 });
```

Figure 17: Requête localhost:3000/api/getBySymbolVolumeTotal/:Symbol

- ✓ **Requête 6** : Pour effectuer le regroupement, nous allons établir le **pipeline** qui est la variable **aggregation**. Dans ce pipeline, nous n'allons filtrer grâce à **\$match**. Nous ne specifions pas de conditions de filtre, dans ce cas tous les documents sont concernés, puis nous regroupons les compagnies en sommant les volumes appartenant à un même

symbole grâce à **\$group** et **\$sum**. Puis, nous utilisons la fonction **aggregate()** qui prendra en argument notre pipeline.

```
80 //Afficher le volume total par symbole
81 route.get('/getByGroupSymbol/', async (req,res)=>{
82     try{
83
84         const aggregation = [
85             {$match : {}},
86             {$group: {_id:"$Symbol", totalVolume:{$sum:"$Volume"}}}
87         ]
88         const dataById = await Model.aggregate(aggregation);
89         res.json(dataById);
90     }catch(error){
91         res.json({message:error.message});
92     }
93 });
```

Figure 18: Requête localhost:3000/api/getByGroupSymbol

- ✓ **Requête 7** : cette requête va nous permettre de mettre à jour un document. Nous définissons une variable **data**. Nous utilisons la fonction **findByIdAndUpdate()** qui permet de retrouver un ou plusieurs documents par l'identifiant(id) et le(s) remplacer par le document déclaré contenu dans la **variable request**.

```
95 //Mise à jour par l'identifiant
96 route.patch('/update/:id', async (req,res)=>{
97     try{
98         const id = req.params.id;
99         const request = {
100             Date: "2022-04-29",
101             Symbole: "ESMT",
102             "Adj Close": 34,
103             Close:35,
104             High:50,
105             Low: 37.456,
106             Open: 45,
107             Volume: 48.555
108         }
109         const update = request;
110         const options = {new:true};
111         const data = await Model.findByIdAndUpdate(id,update,options);
112         res.send(data);
113     }catch(error){
114         res.json({message:error.message});
115     }
116 });
```

Figure 19: Requête localhost:3000/api/update/:id

- ✓ **Requête 8** : cette requête va nous permettre de mettre à jour un document. Nous définissons une variable **data** qui est du même type que les documents de la base de données. Nous utilisons la fonction **updateMany()** qui permet de retrouver un ou plusieurs documents par le Symbol (Symbol) et remplacer le ou les documents par le document déclaré contenu dans la **variable request**.

```
118 //Mise à jour par le nom (Symbole)
119 route.patch('/updateBySymbol/:Symbol', async (req,res)=>{
120     try{
121         const id = req.params.id;
122         const symbole = req.params.Symbol;
123         const request = {
124             Date: "2022-04-29",
125             Symbol: "ESMT",
126             "Adj Close": 34,
127             Close:35,
128             High:50,
129             Low: 37.456,
130             Open: 45,
131             Volume: 48.555
132         }
133         const update = request;
134         const options = {new:true};
135         const data = await Model.updateMany({Symbol:symbole},update,options)
136         res.send(data);
137     }catch(error){
138         res.json({message:error.message});
139     }
140 });
```

Figure 20: Requête localhost:3000/api/updateBySymbol/:Symbol

- ✓ **Requête 9** : Nous utilisons la fonction **findByIdAndDelete(id)**, qui retourne le ou les documents ayant le même id indiqué en argument qui ont été supprimé. On récupère l'id par commande **req.params.id**

```
143 //Supprimer par l'identifiant
144 route.delete('/delete/:id', async (req,res)=>{
145     try{
146         const dataDeleted = await Model.findByIdAndDelete(req.params.id);
147         res.send(dataDeleted);
148     }catch(error){
149         res.status(400).json({message:error.message});
150     }
151 });
```

Figure 21: Requête localhost:3000/api/delete/:id

- ✓ Requête 10 : Nous utilisons la fonction **deleteMany()**, qui retourne le ou les documents ayant le même Symbol indiquer en argument qui ont été supprimé. La fonction permet de retrouver un ou plusieurs documents par le Symbol (Symbol) et de les supprimer.

```
153 //Supprimer tous les documents du symbol
154 route.delete('/deleteBySymbol/:Symbol', async (req,res)=>{
155     try{
156         const symbol = req.params.Symbol
157         const options = {new:true};
158         const dataDeleted = await Model.deleteMany({Symbol:symbol},options);
159         res.send(dataDeleted);
160     }catch(error){
161         res.status(400).json({message:error.message});
162     }
163 })
```

Figure 22: Requête localhost:3000/api/deleteBySymbol/:Symbol

- ✓ Fichier index.js :

Importation des modules : nous allons importer le module express qui est le framework **Node** le plus populaire pour la réalisation d'api ou d'app web. Nous importons également le module **mongoose** qui nous permet de manipuler la base de donnée sur mongoDB, puis on inclut notre fichier de routes. On lance express, en indiquant le port d'écoute (port 3000).

```
1 require('dotenv').config();
2
3 const express = require("express");
4 const mongoose = require("mongoose")
5 const app = express();
6 const routes = require("./routes/routes");
7
8 app.use(express.json())
9 app.use('/api', routes)
10 app.listen(3000, function(){
11     console.log("Server started at port ${3000}");
12 })
```

Figure 23: Importation de module

Database : On récupère le **String de connexion** à la base de donnée dans la variable **mongoString** puis on se connecte à la base grâce à la fonction **connect()**. On utilise la fonction **once()** pour retourner une seule fois le message indiquant qu'on est connecté à la base de donnée et la fonction **on()** pour afficher l'erreur en cas d'erreur de connexion.

```
15  const mongoString = process.env.DATABASE_URL;
16
17  mongoose.connect(mongoString);
18  const database = mongoose.connection;
19
20  database.on('error', (error)=> {
21    |    console.log(error);
22  |  });
23
24  database.once('connected', ()=> {
25    |    console.log("Database connected");
26  |  })
```

Figure 24: Connexion à la Base de Donnée

- **Fichier env:**

Nous avons défini une constante qui récupère les informations de connexion à la base de donnée. Le mot de passe vous a été volontairement caché. Lors d'affectation de valeur, vous devriez écrire en clair votre mot de passe.

```
.env
1  DATABASE_URL = mongodb+srv://EasyTools:<password>@clusterforapi.9buuy.mongodb.net/DB_name
2
```

3.5- Documentation de l'API

Pour la documentation de notre API, veuillez vous rendre au lien ci-dessous.

Lien documentation : <https://documenter.getpostman.com/view/19301807/Uz5CKxfW>

3- Conclusion

L'API estful développé offre de nombreuses fonctionnalités et a été developpé dans un environnement javascript (node js). Des modules importants tel que **dotenv**, **express**, **mongoose** ont été ajouté au projet afin de pouvoir exploiter les fonctions qu'ils offrent et de ce fait de réaliser les fonctionnalités prévues initialement pour l'API. L'API est simple et facile d'usage. La base de données associé est hébergée dans un cluster dans MongoDB Atlas accessible par MongoDB Compass.