Exploration Express.JS 02-Livre d'Or (noSQL)

1 OBJECTIFS

Cette activité doit permettre :

- de pratiquer le framework Express.JS
- d'utiliser un système de base de données en ligne
- de découvrir le système noSQL
- d'aborder le 'pattern design' MVC (Model – View - Controller)

Cette activité est inspirée d'une vidéo sur Youtube :

NET NINJA BLACK-BELT YOUR WEB SKILLS	https://www.youtube.com/watch?v=bxsemcrY4gQ&list=PL4cUxeGkcC9jsz4LDYc6kv3ymONOKxwBU&index=9 Par The Net Ninja

TABLE DES MATIÈRES

2 DÉROULEMENT

Il est obligatoire d'avoir fait l'exploration du livre d'Or MySQL.

La base noSQL est fournie en ligne (user : slam5, pass : koronavirus, base : LivreOr, host : cluster0.ju3uh.mongodb.net)

Il est recommandé de lire l'article suivant :
https://stph.scenari-community.org/contribs/nos/Mongo1/co/sqlComparaison.html

Il faudra supprimer les appels à la base MySQL et les remplacer par des appels à la base noSQL (utilisation de MongoDB et du module mongoose)

2.1 RÉCUPÉRATION DU PROJET LIVRE D'OR

Faire une copie du projet Livre d'Or précédent dans un répertoire "LivreOr noSQL"

supprimer le module mysql

ajouter le module mongoose

3 BASCULE VERS UNE BASE NOSQL

La première découverte est de passer d'une base MySQL a une base noSQL : l'intérêt est de montrer que la transformation est fastidieuse sans une architecture de codage de type MVC.

La contrepartie, c'est que l'apprentissage de noSQL est très simple.

3.1 CORRECTION DE L'ACCÈS (DB.INI)

L'accès se fera sur une base noSQL (de type mongoDB) disponible en ligne.

db.ini

```
[dev]
host = cluster0.ju3uh.mongodb.net
user = slam5
password = koronavirus
dbname = LivreOr
```

3.2 CORRECTION DU CODE PRINCIPAL

Dans le fichier principal, il faut retirer toutes les références à MySQL, mais dans un premier temps, nous allons simplement corriger l'usage du fichier db.ini pour créer le lien URL d'accès à la base.

3.2.1 Création URI

index.js

```
// récupération des paramètres
let configDB = iniparser.parseSync('./DB.ini')
    let host = configDB['dev']['host'];
    let user = configDB['dev']['user'];
    let password = configDB['dev']['password'];
    let database = configDB['dev']['dbname'];

// préparation connexion à la base noSQL
const uri = `mongodb+srv://${user}:${password}@${host}/${database}?
retryWrites=true&w=majority`;
```

Pour rappel, la constante **uri** est construite en utilisant les variables JavaScript dans la chaîne de caractère, grâce à l'utilisation des symboles anticotes '`'.

On conserve l'usage de la section [dev] car il serait possible par la suite de choisir une section [Prod] par exemple.

3.2.2 Lancement application et connexion à la base

Juste en dessous du code précédent, nous allons activer les dépendances et modules pour Express, mais aussi lancer une connexion qui mérite une petite explication.

Index.js

```
// activer les dépendances pour Express et EJS
let app = express()
app.set('view engine', 'ejs')
app.use(express.static('views'))
app.use(express.static('public'))

// activer la connexion asynchrone à la base distante noSQL (promesse)
mongoose.connect(uri, { useNewUrlParser: true, useUnifiedTopology: true})
.then((result) => {
    app.listen(3000, () => console.log(`SRV : le serveur Livre d'Or (noSQL) est prêt.`))
})
.catch((err) => { console.log(err)})
console.log(`DB  : La base de données noSQL distante est prête`)
```

La fonction connect() de Mongoose est une fonction asynchrone : cela signifie que Node.JS va la lancer et passer à la suite du code (console.log) sans attendre le résultat.

Lorsque la connexion sera établie, elle appellera la fonction dans le **.then** ou bien en cas d'erreur, ira dans le **.catch**.

Il s'agit d'une **promesse**.

Le résultat des **console.log()** sera le suivant :

```
DB  : La base de données noSQL distante est prête
SRV : le serveur Livre d'Or (noSQL) est prêt.
```

Cela peut paraître surprenant mais c'est la force principale de JavaScript et Node.JS.

Testez votre code avant de continuer.

3.2.3 Création d'un schéma de données Mongoose

Dans le répertoire 'models', nous allons créer un fichier tlivre.js qui va permettre de décrire les données pour MongoDB.

tlivre.js

```
// Module Mongoose avec son schéma de données
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

// Création du schéma de données
const tlivreSchema = new Schema({
    id: {type: Number, required: true},
    evaluation: {type: String, required: true},
    message: {type: String, required: true},
    name: {type: String, required: true},
}, {timestamps: true});

// création du modèle basé sur le schéma et export de la fonction
const Livre = mongoose.model('Livre', tlivreSchema);
module.exports = Livre;
```

Le format de données dans une base MongoDB est très proche de JSON : ici, chaque attribut a un type et est nécessaire.

3.2.4 Lecture de la base

La déclaration de l'objet qui utilisera notre schéma se fait dans les déclarations de modules :

index.js

```
// inclure les dépendances et middlewares
...
const mongoose = require('mongoose')
const Livre = require('./models/tlivre')
```

La lecture de la base se fait par l'usage de la fonction objetBase.find() qui est asynchrone. Il faut donc de nouveau utiliser les promesses.

index.js

```
// voir tous les messages
app.get('/LivreOr', (req, res) => {
    Livre.find()
    .then((livres) => {
        console.log(livres)
        res.render('affichage', {livres})
    })
    .catch((err) => {
        console.log(err)
    })
})
```

Il faut donc modifier la vue, pour lire les données de l'objet 'livres'.

affichage.ejs

```
<%- include('header') %>
  <h1>Affichage : </h1>
  <div class="ui main container">
     <ul> <% livres.forEach(function(user){ %>
         <li>
            <a href="http://localhost:3000/LivreOr/<%=user.id%>" >&#128065;</a>
            <a href="http://localhost:3000/LivreOr/update/<%=user.id%>"
>&#9998;</a>
            <a href="http://localhost:3000/LivreOr/delete/<%=user.id%>"
>&#10006;</a>
            <strong> <%= user.name %> </strong> a dit <%= user.message %>
         </li>
         <% })%>
      </ul>
  </div>
<%- include('footer') %>
```

Testez votre code.

3.2.5 Écriture dans la base

Il suffit de modifier la fonction qui appelle l'écriture (.post('/LivreOr')

index.js

```
// DONE: réception formulaire urlEncoded... changer le formulaire : application/x-www-form-urlencoded
.post('/LivreOr', (req, res) => {
    let msgID = req.body.id
    let msgName = req.body.name
    let msgMsg = req.body.msg
    let msgNote = req.body.note

    console.log(`Ajout msg ID ${msgID} de ${msgName} contenant ${msgMsg} et noté ${msgNote}`)
    // Création d'un nouvel objet commentaire Livre
    const commentLivre = new Livre(
        {
            id: msgID,
            evaluation: msgNote,
            message: msgMsg,
            name: msgName
        }
    );
    commentLivre.save()
    .then((result) => {
        console.log("Insertion terminé");
        res.redirect("/LivreOr");
    })
    .catch((err) => {
        console.log(err)
    })
})
```

La requête n'est absolument pas comparable à une requête SQL : ce type de base porte donc bien son nom de 'noSQL'.

3.2.6 Erreur 404

Au passage, on peut ajouter ce code de **middleware**, complètement à la fin :

```
// erreur 404
.use((req, res) => {
    res.status(404).render('404nontrouve')
});
```

À vous de créer la page EJS d'erreur 404.

4 APPLICATION ET SOUS-FONCTIONS

Dans cette partie, nous allons nettoyer le projet : en grossissant, le code devient moins lisible et surtout, un seul fichier contient l'essentiel du programme.

- Le travail en équipe est plus compliqué (tout le monde intervient sur le même fichier)
- La sécurité n'est pas assurée (quelqu'un peu modifier une partie qui ne le concerne pas, par inadvertance)
- L'application n'est pas modulaire

L'objectif est de classer le code par fonctions générales. Il faut nettoyer les routes en regroupant les routes communes.

4.1 APPLICATION GESTION DES ROUTES

Bien que dans notre livre d'Or, nous n'avons qu'une seule route principale /LivreOr, nous allons déplacer les routes concernées dans un fichier prévu à cet effet, dans le répertoire 'routes' à la racine du projet.

4.1.1 Nouveau fichier LivreRoute.js

Dans ce fichier (LivreRoute.js) nous allons créer un nouveau routeur Express ('routeur') puis coller toutes les routes pointant sur /LivreOr. Il faut ensuite changer 'app' pour 'routeur' (Alt + double-clic dans VSCode est pratique). Let's go !

LivreRoute.js

```
// création du routeur Express pour ce module
const express = require('express');
const routeur = express.Router();

// voir tous les messages
routeur.get('/LivreOr', (req, res) => {
    Livre.find()
    .then((livres) => {
        console.log(livres)
        res.render('affichage', {livres})
    })
    .catch((err) => {
        console.log(err)
    })
})

// chercher les messages contenant ElementSearch dans le champ message (query)
routeur.get('/search/', (req, res) => {
    let critere = '%'+req.query.msgSearch+'%'   //query. Exemple : %you%
    console.log("Find this : "+critere)
    /*
    mysqlconnexion.query('SELECT * FROM tlivre WHERE message LIKE ?', [critere], (err, lignes,
champs) => {
        if (!err) {
            console.log(lignes)
            res.send(lignes)
        }
    })
    */
})

// voir un message choisi par ID (parameters)
routeur.get('/LivreOr/:id', (req, res) => {
    let critere = req.params.id;
    console.log("ID = "+critere);
    Livre.findOne({id: critere})
    .then((livres) => {
        console.log(livres);
        res.send(livres);
    })
    .catch((err) => {
        console.log("Oups! ",err);
    });

})
routeur.get('/formulaire', (req, res) => {
   res.render('./formulaire')
```

```
})

// effacer un message choisi par ID (parameters)
routeur.get('/LivreOr/delete/:id', (req, res) => {
    let critere = req.params.id
    console.log("ID = "+critere)
    Livre.findOneAndDelete({id: critere})
    .then((livres) => {
        console.log(livres);
        res.redirect("/LivreOr");
    })
    .catch((err) => {
        console.log("Oups! ",err);
    });
})

// ajouter un message (query)
.post('/LivreOr', (req, res) => {
    let msgID = req.body.id
    let msgName = req.body.name
    let msgMsg = req.body.msg
    let msgNote = req.body.note

    console.log(`Ajout msg ID ${msgID} de ${msgName} contenant ${msgMsg} et noté ${msgNote}`)
    const commentLivre = new Livre(
        {
            id: msgID,
            evaluation: msgNote,
            message: msgMsg,
            name: msgName
        }
    );
    commentLivre.save()
    .then((result) => {
        console.log("Insertion terminé");
        res.redirect("/LivreOr");
    })
    .catch((err) => {
        console.log(err)
    })
})
```

Il faut enfin déclarer ce fichier comme étant un module pour le fichier principal, en écrivant à la fin :

LivreRoute.js

```
module.exports = routeur ;
```

4.1.2 Modifications des fichiers existants

Nous déplaçons aussi les routes (/formulaire et /search) qui ne sont pas dans /LivreOr mais qui le concerne également. Cela permet de simplifier les routes dans le fichier LivreRoute.js

Header.ejs

```
...
        <div class="ui inverted menu">
            <a href="/LivreOr/formulaire" class="header item">Formulaire</a>
            <a href="/LivreOr" class="header item">Résultats</a>
        </div>
…
```

Il faut déclarer notre nouveau module, au début du fichier index.js :

index.js

```
…
const mongoose = require('mongoose')
const Livre = require('./models/tlivre')
const Routeur = require('./routes/LivreRoute');
…
```

Il suffit maintenant d'appeler notre route, avant la page non trouvée dans le fichier index.js. Nous pourrions utiliser la commande app.use(Routeur) mais ici, nous allons déjà filtrer les requêtes en direction de /LivreOr :

index.js

```
app.use('/LivreOr', Routeur);
```

Grâce à cela, il n'est plus nécessaire d'inclure la route complète dans LivreRoute.js mais seulement ce qui suit /LivreOr.

Cela permet de créer des applications modulaires, et s'il faut changer le chemin d'accès, il n'y a que la ligne dans index.js à modifier.

exemple LivreRoute.js

```
// voir tous les messages
routeur.get('/~~LivreOr~~', (req, res) => {
```

Comme il n'est donc plus nécessaire de tester la partie de chemin /LivreOr dans LivreRoute, il faut modifier les routes dans LivreRoute.js.

Les routes dans ce fichier sont les suivantes :

LivreRoute.js

```
// voir tous les messages ( / devient la racine de /LivreOr )
routeur.get('/', (req, res) => {
...

// insérer une nouvelle valeur
routeur.get('/formulaire', (req, res) => {
...

// chercher les messages contenant ElementSearch dans le champ message (query)
routeur.get('/search', (req, res) => {
...

// voir un message choisi par ID (parameters)
routeur.get('/:id', (req, res) => {
...

// effacer un message choisi par ID (parameters)
routeur.get('/delete/:id', (req, res) => {
...

// ajouter un message (query)
.post('/', (req, res) => {
...
```

Respectez bien cet ordre, sinon, l'accès à formulaire sera analysé comme un ":id" : le fichier est en effet parcouru de haut en bas, la première occurrence de routage sera choisie et l'instruction res.render() empêche de continuer le parcours.

Notez également qu'il faut remettre **app** devant le .use(...)) ; dans le fichier index.js.

4.1.3 Test de fonctionnement

Malgré cela, le programme devrait planter avec le message suivant :

```
ReferenceError: Livre is not defined
    at D:\david\WorkSpaces\NodeJS projects\LivreOr2 noSQL\routes\LivreRoute.js:7:5
```

En effet, les constantes déclarées dans le fichier index.js ne sont pas visibles dans le fichier LivreRoutes.js

4.1.4 Déplacement du modèle

Il faut donc supprimer la ligne de déclaration de Livre dans index.js et la modifier comme suit dans LivreRoute.js

LivreRoute.js

```
const Livre = require('./models/tlivre');
```

(le double point est nécessaire, car LivreRoute.js est dans le répertoire /routes)

Cette fois, l'application doit fonctionner normalement.

5 Design Pattern MVC

Nous allons en profiter pour rendre le code plus propre en séparant nos fichiers, en ajoutant un répertoire 'models'. C'est une approche dite "MVC" pour indiquer qu'on sépare les modèles de données (les accès aux données), les différentes vues (répertoire 'views') et éventuellement les contrôleurs qui gèrent les actions.

Le code pour les vues est déjà placé dans le répertoire '**views**' et est appelé grâce à la fonction render() d'express (et réalisé par le moteur EJS).

Le code pour les modèles sera dans le répertoire existant '**models**', utilisé pour le schéma de données de Mongoose.

Il faut donc créer un répertoire '**controllers**', ou nous placerons les fonctions appelées lors du routage.

5.1.1 Exportation des contrôleurs

Dans le code ci-dessous, la partie en italique orange représente le contrôleur, c'est-à-dire les décisions prise pour la route '/'.

```
// voir tous les messages
routeur.get('/', (req, res) => {
    Livre.find()
    .then((livres) => {
        console.log(livres, "------------------------")
        res.render('affichage', {livres})
    })
    .catch((err) => {
        console.log(err)
    })
});
```

L'idée est de déplacer ce code, dans le répertoire 'controllers' : nous allons créer un fichier LivreController.js.

Ce fichier doit contenir la déclaration des modules utilisés, notamment, le modèle Livre :

LivreController.js

```
// Les actions sur le LivreOr
// convention d'écriture : livreOr_affichage, livreOr_creation,
livreOr_suppression...

// déclaration du modèle Mongoose utilisé ici
const Livre = require('../models/tlivre');

const livreOr_affichage =
```

Il suffit de copier le code italique orange après le signe égal :

LivreController.js

```
const livreOr_affichage = (req, res) => {
    Livre.find()
    .then((livres) => {
        console.log(livres, "------------------------")
        res.render('affichage', {livres})
    })
    .catch((err) => {
        console.log(err)
    })
}
```

Faire de même pour toutes les fonctions des différentes routes.

Il faut maintenant exporter les fonctions depuis LivreController.js vers index.js :

LivreController.js

```
module.exports = {
    livreOr_affichage,
    livreOr_creation,
    livreOr_suppression
}
```

Il s'agit d'une liste d'objet en JavaScript, dont l'appel se fera dans LivreRoute.js

LivreRoute.js

```
const LivreControle = require('../controllers/LivreController');
...
// voir tous les messages
routeur.get('/', LivreControle.livreOr_affichage);

// insérer une nouvelle valeur
routeur.get('/formulaire', LivreControle.livreOr_creation);
```

Faire de même pour toutes les autres fonctions déclarées dans LivreController.js

5.1.2 Contrôleur final

Après une petite mise en forme, voici à quoi correspond le fichier des routes pour /LivreOr :

LivreRoute.js

```
// création du routeur Express pour ce module
const express = require('express');
const routeur  = express.Router();
const LivreControle = require('../controllers/LivreController');

// voir tous les messages
routeur.get('/', LivreControle.livreOr_affichage)
        .get('/formulaire', LivreControle.livreOr_creation)
        .get('/search', LivreControle.livreOr_recherche)
        .get('/:id', LivreControle.livreOr_selection)
        .get('/delete/:id', LivreControle.livreOr_suppression)
        .post('/', LivreControle.livreOr_ecriture);

module.exports = routeur;
```

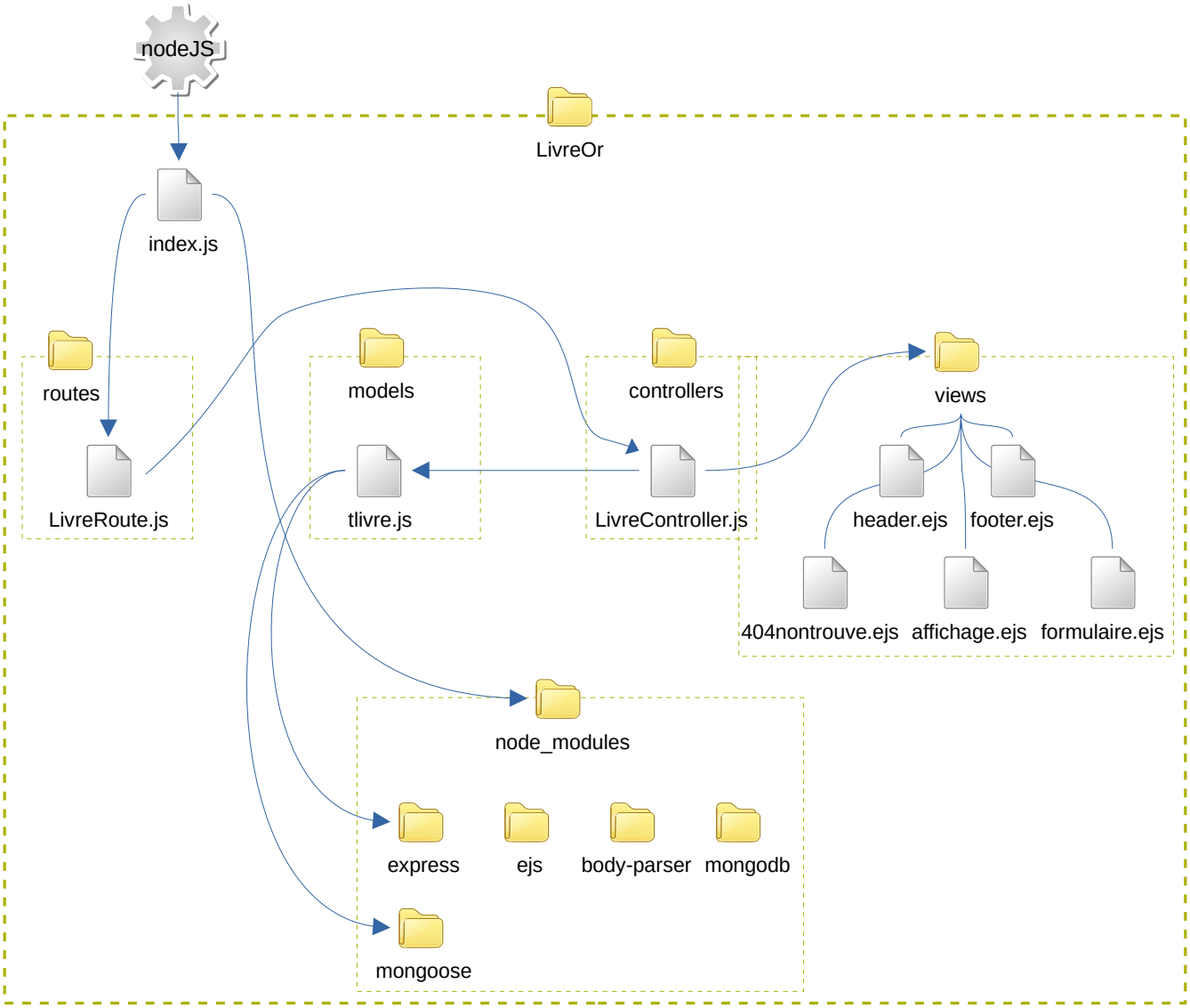
C'est un code élégant, facile à modifier. Ajouter une action ne prend qu'une ligne. Concernant le contrôleur :

```
// déclaration du modèle Mongoose utilisé ici
const Livre = require('../models/tlivre');

// Les actions sur le LivreOr
const livreOr_affichage = (req, res) => {···
}

const livreOr_creation = (req, res) => {···
}

const livreOr_ecriture = (req, res) => {···
}

const livreOr_suppression = (req, res) => {···
}

const livreOr_selection = (req, res) => {···
}

const livreOr_recherche = (req, res) => {···
}

module.exports = {
    livreOr_affichage,
    livreOr_creation,
    livreOr_ecriture,
    livreOr_suppression,
    livreOr_selection,
    livreOr_recherche
}
```

C'est également un code facile à maintenir.

6 VUE GLOBALE DE L'APPLICATION

Pour éclaircir les idées sur ce cours, voici comment l'application fonctionne désormais :



Dans sa nouvelle forme, il devient facile de créer une gestion des utilisateurs pouvant enregistrer des commentaires, ou bien de vérifier (cookie de session) les droits d'un utilisateur.

Les améliorations sont la **gestion des erreurs** (remplacer les console.log() par des vues), **l'authentification** (création d'une route et des éléments utiles), des **filtres** (vues des meilleurs commentaires), etc.

Désormais, vous avez les éléments pour travailler sur un petit projet réel.

7 ANNEXES

7.1 CRÉATION D'UN COMPTE NOSQL SUR NOSQL ATLAS

Ce n'est pas nécessaire pour cette activité, mais vous pourriez vouloir créer une application personnelle partagée.

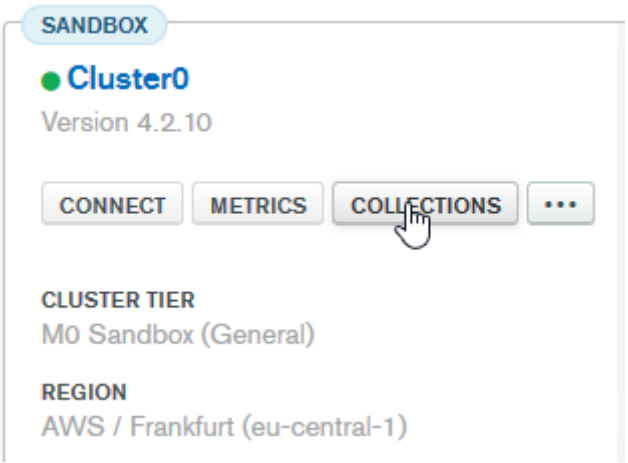
Se rendre sur : https://www.mongodb.com/cloud/atlas/register

remplir les champs.

- Créer une organisation (anciennement, un groupe)
- Créer un nouveau projet
- Créer un cluster

Tier	RAM	Storage	vCPU	Base Price
M0 Sandbox	Shared	512 MB	Shared	Free forever
M0 clusters are best for getting started, and are not suitable for production environments.				
500 max connections | Low network performance | 100 max databases | 500 max collections				
M2	Shared	2 GB	Shared	$9 / MONTH
M5	Shared	5 GB	Shared	$25 / MONTH

Une fois le cluster créé, il faut créer une nouvelle collection (équivalent d'une base sous MySQL) :



Choisir [Add my own datas],

Donner un Nom à la base et à la "table",

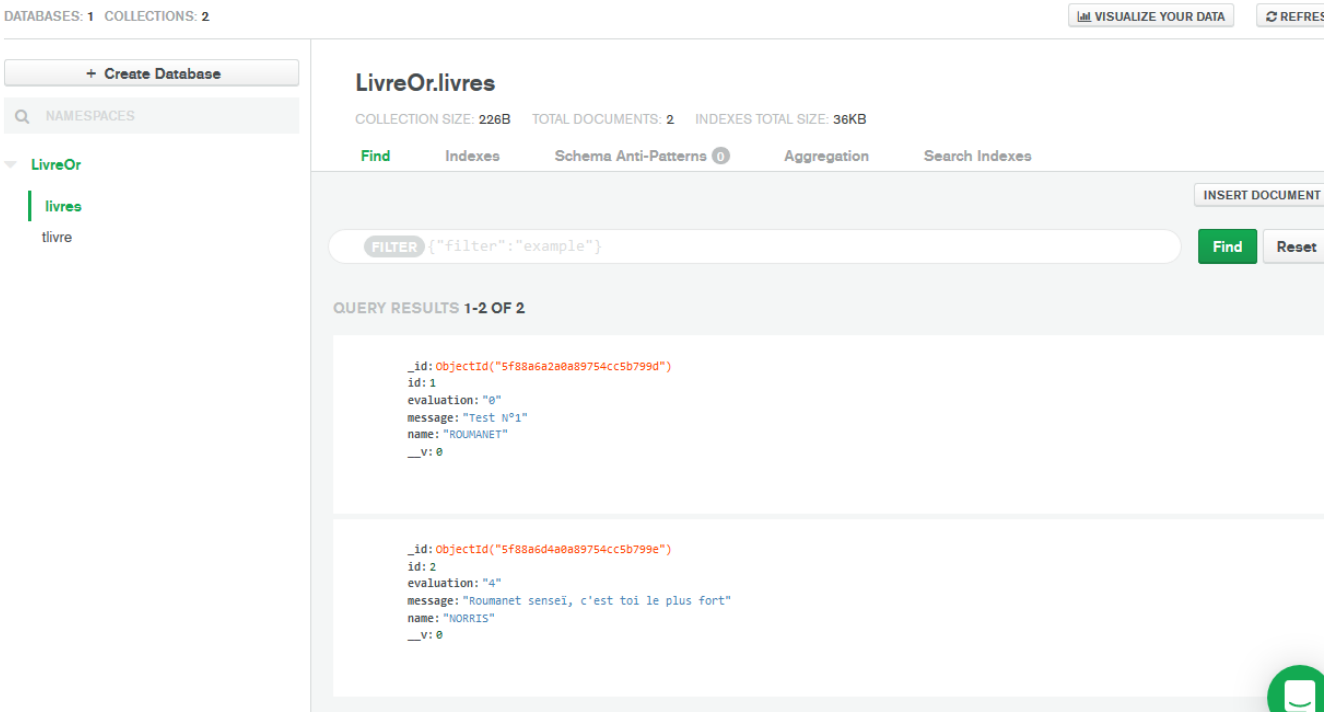
Créer un utilisateur capable de lire et écrire dans cette base (slam5 / koronavirus),

En cliquant sur [Connect] il faut ensuite définir les IP qui pourront se connecter et récupérer l'URL de connexion.

7.2 Résultat et données dans la base MongoDB

Vous pouvez vérifier le résultat de vos écritures dans la base de votre création (si vous utilisez la base fournie pour cette exploration, vous ne pourrez pas y accéder vous même)

7.2.1 Vue générale :



7.2.2 Détails :

```
{
  _id: 5f902a638799e829649b9498,
  id: 15,
  evaluation: '2',
  message: 'Test MVC avec controllers OK',
  name: 'ROUMANET',
  createdAt: 2020-10-21T12:32:35.257Z,
  updatedAt: 2020-10-21T12:32:35.257Z,
  __v: 0
}
```

Vous pouvez constater qu'un ID propre à MongoDB est créé, sous le nom de '_id'