MONGODB

INTRODUCTION

- Une base de données SQL est une base de données **relationnelle** qui emploie le langage **S**tructured **Q**uery **L**anguage pour gérer les données qu'elle contient. Généralement, une base de données SQL organise ses données dans des **tables** selon des **schémas** stricts.

- MongoDB est une base de données **NoSQL**. Les données sont stockées comme des **collections** de **documents** individuels décrits en JSON (JavaScript Object Notation). Il n'y a pas de schéma strict de données, et il n'y a pas de relation concrète entre les différentes données.

- Les avantages principaux de MongoDB sont son **évolutivité** et sa **flexibilité**.

CONNECTION API - CLUSTER MONGODB

installation Mongoose : npm install mongoose

Mongoose est un package qui facilite les interactions avec la bdd MongoDB. Il permet de :

- **valider** le format des données ;
- gérer les **relations** entre les documents ;
- **communiquer** directement avec la bdd pour la lecture et l'écriture des documents

EXEMPLE CODE

```
mongoose.connect(process.env.MONGODB_URI,
  { useNewUrlParser: true,
    useUnifiedTopology: true })
  .then(() => console.log('Connexion à MongoDB réussie !'))
  .catch(() => console.log('Connexion à MongoDB échouée !'));
```

CRÉATION D'UN SCHÉMA THING

- Exemple : créer un dossier appelé models où il se trouve un fichier appelé thing.js :

```
const mongoose = require('mongoose');
const thingSchema = mongoose.Schema({
  title: { type: String, required: true },
  description: { type: String, required: true },
  imageUrl: { type: String, required: true },
  userId: { type: String, required: true },
  price: { type: Number, required: true },
});
module.exports = mongoose.model('Thing', thingSchema);
```

ENREGISTREMENT DES THINGS DANS LA BDD

- Pour pouvoir utiliser le modèle Mongoose dans l'application, on doit l'importer dans le fichier app.js

```
const Thing = require('./models/thing');
```

- Maintenant, on remplace la logique de la route POST par :

```
app.post('/api/back', (req, res, next) => {
  delete req.body._id;
  const thing = new Thing({
    ...req.body
  });
  thing.save()
    .then(() => res.status(201).json({ message: 'Objet enregistré !'}))
    .catch(error => res.status(400).json({ error }));
});
```

EXPLICATION :

Ici, une instance du modèle Thing est crée en lui passant un objet JavaScript contenant toutes les informations requises du corps de requête analysé (en ayant supprimé en amont le faux_id envoyé par le front-end).

L'opérateur spread ... est utilisé pour faire une copie de tous les éléments de req.body.

Ce modèle comporte une méthode save() qui enregistre simplement le Thing dans la base de données.
*La base de données MongoDB est fractionnée en **collections** : le nom de la collection est défini par défaut sur le pluriel du nom du modèle. Ici, ce sera **Things** .*

La méthode save() renvoie une Promise :
Dans la bloc then() : renverrons une réponse de réussite avec un code 201 de réussite.
Dans notre bloc catch() : renverrons une réponse avec l'erreur générée par Mongoose ainsi qu'un code d'erreur 400.

RÉCUPÉRATION DE LA LISTE DE THINGS EN VENTE

- On peut implémenter la route GET afin qu'elle renvoie tous les Things dans la bdd :

```
app.use('/api/back', (req, res, next) => {
  Thing.find()
    .then(things => res.status(200).json(things))
    .catch(error => res.status(400).json({ error }));
});
```

- On utilise la méthode find() dans le modèle Mongoose afin de renvoyer un tableau contenant tous les Things dans la base de données. À présent, si on veut ajouter un Thing , il doit s'afficher immédiatement sur la page d'articles en vente.
- En revanche, si on clique sur l'un des Things , l'affichage d'un seul élément ne fonctionne pas. En effet, il tente d'effectuer un appel GET différent pour trouver un Thing individuel. Implémentons cette route maintenant.

RÉCUPÉRATION D'UN THING SPÉCIFIQUE

- Ajoutons une autre route à notre application, juste après notre route POST :

```
app.get('/api/back/:id', (req, res, next) => {
  Thing.findOne({ _id: req.params.id })
    .then(thing => res.status(200).json(thing))
    .catch(error => res.status(404).json({ error }));
});
```

On utilise :

- la méthode get() pour répondre uniquement aux demandes GET à cet endpoint ;
- deux-points : en face du segment dynamique de la route pour la rendre accessible en tant que paramètre ;
- la méthode findOne() dans le modèle Thing pour trouver le Thing unique ayant le même _id que le paramètre de la requête ;

RESULTAT

- L’ application commence vraiment à prendre forme

- On peut créer des objets et les voir apparaître immédiatement dans la boutique en ligne grâce à la base de données. Et même ouvrir un objet en particulier pour obtenir les informations de cet objet précis, via la base de données.

MIS À JOUR D'UN THING EXISTANT

- Ajoutons une autre route à notre application, juste en dessous de la route GET individuelle. Cette fois, elle répondra aux requêtes PUT :

```
app.put('/api/back/:id', (req, res, next) => {
  Thing.updateOne({ _id: req.params.id }, { ...req.body, _id: req.params.id })
    .then(() => res.status(200).json({ message: 'Objet modifié !'}))
    .catch(error => res.status(400).json({ error }));
});
```

- on exploite la méthode updateOne() dans le modèle Thing . Cela permet de mettre à jour le Thing qui correspond à l'objet que nous passons comme premier argument.
- on utilise le paramètre id passé dans la demande, et le remplaçons par le Thing passé comme second argument.

RESULTAT

- on peut tester la nouvelle route en cliquant sur un Thing de l'application, puis sur son bouton Modifier, changer les paramètres et sauvegarder.

- Le Thing modifié sera alors envoyé au back-end.

- En revenant sur la page des articles, on devra retrouver l’article modifié.

SUPPRESSION D'UN THING

- Il est temps d'ajouter une dernière route, la route DELETE :

```
app.delete('/api/back/:id', (req, res, next) => {
  Thing.deleteOne({ _id: req.params.id })
    .then(() => res.status(200).json({ message: 'Objet supprimé !'}))
    .catch(error => res.status(400).json({ error }));
});
```

- La méthode deleteOne() fonctionne comme findOne() et updateOne() dans le sens où lui passons un objet correspondant au document à supprimer. On envoie ensuite une réponse de réussite ou d'échec au front-end.