Sequelize

Sequelize
Partie 1

NODE.JS ET LES BASES DE DONNÉES

- Comme dans tout écosystème de développement, il existe différentes manière de travailler avec des bases de données avec Node.JS
- Nous allons travailler avec MySQL, histoire de rester dans un périmètre connu
- Node.JS dispose de plusieurs bibliothèques permettant de travailler avec MySQL.

- Néanmoins, j'ai eu envie pour vous (et pour moi aussi) de travailler avec un ORM assez connu : sequelize.
- ORM cela veut dire Object Relational Model en "français", on parle de Mapping Objet Relationnel.
- Le principe est simple :
  - Beaucoup de requêtes SQL de type CRUD sont assez mécaniques et peuvent être générées automatiquement.
  - Utilisation de classes comme modèles pour accéder aux données d'une table.
  - Et bien d'autres choses.

ORM

- Avantage :
  - Réduire la quantité de code à écrire (pas sur des exemples)
  - Efficace sur des cas simples
  - Abstraction des différents types de base de données
  - Uniformisation du code

- Inconvénients :
  - Coût d'une couche logicielle supplémentaire
  - Ne produit pas toujours des requêtes optimisées
  - Nécessite un apprentissage particulier
  - Peut être inutile sur des requêtes complexes

SEQUELIZE

- ORM basé sur les promesses JavaScript
- Open source, licence MIT
- Actuellement en version 6.x

- Peut travailler avec :
  - Postgres
  - MySQL
  - MariaDB
  - SQLite
  - SQL Server

PRÉPARATION DU PROJET

- Créer une base de données `orm-demo` sous MySQL

- Créer un projet `sequelize-demo`
  - Faire `npm init -y`
  - Installer `mysql2` et `sequelize : npm i mysql2 sequelize`
  - Installer `nodemon` en devDependencies et configurer le script start
  - Créer un fichier `index.js`
  - Lancer `npm start`

CONFIG.JS

- Créer un fichier `/config.js`
- L'adapter à vos identifiants de connexion à MySQL

/config.js

```
1   module.exports = {
2       HOST : 'localhost',
3       USER : 'web',        // Modifier selon votre BDD
4       PASSWORD : 'web',    // Modifier selon votre BDD
5       DB :'orm-demo',
6       dialect: 'mysql',
7       pool : {
8           max : 5,
9           min : 0,
10          acquire : 30000,
11          idle : 10000
12      }
13  }
```

- Ligne 7 : ce sont les paramètres du pool de connexions (5 connexions maximum, un timeout de 30 secondes pour signifier un échec de connexion et une libération automatique de la connexion au bout de 10 secondes d'inactivité).

CONNEXION

- Editer le fichier `/index.js`

/index.js

```
1   const dbConfig = require('./config');
2
3   const { Sequelize, Op, DataTypes, QueryTypes } = require('sequelize');
4   const sequelize = new Sequelize(
5       dbConfig.DB, dbConfig.USER, dbConfig.PASSWORD, {
6           host : dbConfig.HOST,
7           dialect : dbConfig.dialect,
8           pool : {
9               max : dbConfig.pool.max,
10              min : dbConfig.pool.min,
11              acquire : dbConfig.pool.acquire,
12              idle : dbConfig.pool.idle
13          },
14          query : {
15              raw : true
16          }
17      }
18  );
```

- On retrouve les informations de connexion utilisées, celle du pool.
- Ligne 15 : permet de récupérer systématiquement les données brutes (sans ajout de la part de Sequelize).

UN PREMIER MODÈLE

- Créer le fichier `/profsModel.js`

/profsModel.js

```
1   module.exports = (sequelize, DataTypes) => {
2       return sequelize.define('profs', {
3           nom : {
4               type : DataTypes.STRING
5           },
6           prenom : {
7               type : DataTypes.STRING
8           },
9           bureau : {
10              type : DataTypes.INTEGER
11          }
12      })
13  }
```

- Sequelize va automatiquement ajouter (possibilité d'indication contraire) :
  - Une clé primaire, entier, autoincrémentée
  - Un DateTime de création : `createdAt`
  - Un DateTime de mise à jour : `updatedAt`
  - Un 's' au nom de la table, sauf s'il en a déjà un

SYNCHRONISATION

- Dans le fichier `/index.js` ajouter les lignes suivantes

/index.js

```
20   const profsModel = require('./profsModel')(sequelize,DataTypes);
21
22   sequelize.sync();
```

- Ça y est, nous sommes prêts à travailler avec notre modèle.
- Ligne 22 : cette commande va automatiquement lancer une synchronisation des modèle qui réalise un `CREATE TABLE IF NOT EXISTS`

```
[nodemon] starting `node index.js`
Executing (default): CREATE TABLE IF NOT EXISTS `profs` (`id` INTEGER NOT NULL auto_increment , `n
om` VARCHAR(255), `prenom` VARCHAR(255), `bureau` INTEGER, `createdAt` DATETIME NOT NULL, `updated
At` DATETIME NOT NULL, PRIMARY KEY (`id`)) ENGINE=InnoDB;
Executing (default): SHOW INDEX FROM `profs`
[nodemon] clean exit - waiting for changes before restart
```

SYNCHRONISATION

- En ces quelques pages, on se rend vite compte que sequelize fait beaucoup de choses automatiquement.
- Normal c'est un peu son but.
- On peut toutefois légitimement s'interroger sur la nécessité de faire cette synchronisation systématiquement. On la trouve souvent dans les exemples et tutoriels, pour ne pas dire systématiquement.
- Je n'ai pas de réponse franche à ce questionnement.

- L'ajout d'un modèle, lancerait automatiquement la création de la table.
- La modification d'un modèle ne serait pas pris en compte.

- Il est possible de faire une synchronisation unitaire d'un modèle.
- Il est aussi possible de forcer la synchronisation (suppression puis création de la table).

SYNCHRONISATION

- Ajout de modèle, modification de modèle, ce sont des choses que l'on fait plutôt dans les phases de développement.
- Sequelize possède une interface CLI qui permet de scripter des modifications de cet ordre, faire des migrations de modèle et bien d'autres choses.

- Bref, nous nous contenterons de faire cette synchronisation systématiquement dans les exemples qui vont suivre.

Sequelize
Partie 2

CRÉER UN PROF

- Première étape d'un CRUD : le create.
- Ajouter ces lignes au fichier `/index.js`

/index.js

```
25  profsModel.create({
26      nom : "Gaudin",
27      prenom : "Hervé",
28      bureau : 217
29  })
30  .then( data => {
31      console.log("Prof créé : ",data.dataValues);
32  })
33  .catch( err => {
34      console.log("Erreur ",err.message);
35  })
```

- Ça se comprend tout seul.

```
Prof créé :  {
  id: 1,
  nom: 'Gaudin',
  prenom: 'Hervé',
  bureau: 217,
  updatedAt: 2020-05-04T21:01:00.047Z,
  createdAt: 2020-05-04T21:01:00.047Z
}
```

CRÉER UN PROF

- Attention : ré-exécuter le code tel quel rajoutera le même prof.

- Créer les enseignants suivants :
  - Ruchaud William 210
  - Oulad Moussa Mustapha 213
  - Chervy Benjamin 218

RÉCUPÉRER TOUS LES PROFS

- Il suffit de faire un `findAll()`.
- Penser à mettre en commentaire le/les `create()`.

/index.js

```
25  profsModel.findAll()
26  .then(data => {
27      let retour = data;
28      console.log(retour);
29  })
30  .catch( err => {
31      console.log("Erreur : ",err.message)
32  })
```

RÉCUPÉRER LE NOM DE CERTAINS PROFS

- `findAll()` accepte des paramètres comme `attributes` pour filtrer les colonnes et `where` pour restreindre la sélection.
- Nous voulons les profs dont le bureau est > 215

/index.js

```
25  profsModel.findAll({
26      attributes: ['nom'],
27      where : { bureau: {[Op.gt]: 215}}
28  })
29  .then(data => {
30      let retour = data;
31      console.log(retour);
32  })
33  .catch( err => {
34      console.log("Erreur : ",err.message)
35  })
```

```
[ { nom: 'Gaudin' }, { nom: 'Chervy' } ]
```

- `Op.gt` ligne 27 est la représentation de >

RECHERCHER UN PROF PAR SON ID

- `findByPk()` est faite pour ça.

/index.js

```
25  profsModel.findByPk(2)
26  .then(data => {
27      let retour = data;
28      console.log(retour);
29  })
30  .catch( err => {
31      console.log("Erreur : ",err.message)
32  })
```

```
{
  id: 2,
  nom: 'Ruchaud',
  prenom: 'William',
  bureau: 210,
  createdAt: 2020-05-04T21:05:04.000Z,
  updatedAt: 2020-05-04T21:05:04.000Z
}
```

RECHERCHER UN PROF PAR SON ID

- Même requête que la précédente avec l'option `raw : false` dans la configuration (ligne 15 en principe).
- Ce n'est qu'un extrait… cela fait 54 lignes.

```
profs {
  dataValues: {
    id: 2,
    nom: 'Ruchaud',
    prenom: 'William',
    bureau: 210,
    createdAt: 2020-05-04T21:05:04.000Z,
    updatedAt: 2020-05-04T21:05:04.000Z
  },
  _previousDataValues: {
    id: 2,
    nom: 'Ruchaud',
    prenom: 'William',
    bureau: 210,
    createdAt: 2020-05-04T21:05:04.000Z,
    updatedAt: 2020-05-04T21:05:04.000Z
  },
  _changed: {},
  _modelOptions: {
    timestamps: true,
    validate: {},
```

AUTRES MÉTHODES FIND

- Il existe d'autres méthodes find :
  - `findOne()` : ne retourne que le premier résultat
  - `findOrCreate()` : créer une ligne sauf si elle existe déjà
  - `findAndCountAll()` : pour récupérer directement le nombre de lignes.

FAIRE UN UPDATE

- Il suffit de passer les colonnes à modifier et la condition `where`.
- Attention cette opération retourne le nombre de lignes affectées.

/index.js

```
25  profsModel.update(
26      { bureau : 217},
27      { where : { nom : "Ruchaud"} }
28  )
29  .then(num => {
30      console.log(num);
31  })
32  .catch( err => {
33      console.log("Erreur : ",err.message)
34  })
```

FAIRE UNE SUPPRESSION

- Il suffit de passer les colonnes à modifier et la condition `where`.
- Attention cette opération retourne le nombre de lignes affectées.

/index.js

```
25  profsModel.destroy(
26      { where : { nom : "Ruchaud"} }
27  )
28  .then(num => {
29      console.log(num);
30  })
31  .catch( err => {
32      console.log("Erreur : ",err.message)
33  })
```

EXÉCUTER DU SQL

- Il faut rajouter `QueryTypes` dans le `require` de sequelize

```
3   const { Sequelize, Op, DataTypes, QueryTypes } = require('sequelize');
```

/index.js

```
25  sequelize.query("INSERT INTO profs (nom,prenom,bureau,createdAt,updatedAt) VALUES
        (:n,:p,:b,:c,:u)",
26  {
27      replacements: {
28          n : "Ruchaud",
29          p : "William",
30          b : 210,
31          c : '2020-05-04 23:44:00',
32          u : '2020-05-04 23:44:00'
33      },
34      type : QueryTypes.INSERT
35  });
```

CONCLUSION

- Ce n'est évidemment qu'un tout petit aperçu des possibilités de Sequelize.
- Il y a bien d'autres possibilités.
- La documentation officielle et les différents tutoriels vous aideront à trouver ce qui ne pourrait trouver place dans ce support de cours.

- Nous allons maintenant pouvoir faire fonctionner notre API REST sur les enseignants avec Sequelize.

CONFIG.JS

- Créer un fichier `/config.js` pour les paramètres de connexion.
- Adapter le fichier à votre contexte.

/config.js

```
1   module.exports = {
2       HOST : 'localhost',
3       USER : 'web',
4       PASSWORD : 'web',
5       DB : 'api-profs',
6       dialect : 'mysql',
7       pool : {
8           max : 5,
9           min : 0,
10          acquire : 30000,
11          idle : 10000
12      }
13  }
```

LES MODELES

- Créer un dossier `/models` pour y placer les modèles
- Créer le fichier `/models/profsModel.js`.

/models/profsModel.js

```
1   module.exports = (sequelize,DataTypes) => {
2       return sequelize.define('profs', {
3           nom: {
4               type : DataTypes.STRING
5           },
6           prenom: {
7               type : DataTypes.STRING
8           },
9           bureau: {
10              type : DataTypes.INTEGER
11          }
12      })
13  }
```

- Bien veiller à ce qu'il n'y ait pas de ligne tout en haut du code, Visual Studio Code se croit parfois obligé d'en rajouter.

LES MODELES

- Créer le fichier `/models/index.js`.

/models/index.js

```
1   const dbConfig = require('../config');
2
3   const { Sequelize, DataTypes } = require('sequelize');
4   const sequelize = new Sequelize(
5       dbConfig.DB, dbConfig.USER, dbConfig.PASSWORD, {
6           host : dbConfig.HOST,
7           dialect : dbConfig.dialect,
8           pool: {
9               max : dbConfig.pool.max,
10              min : dbConfig.pool.min,
11              acquire : dbConfig.pool.acquire,
12              idle : dbConfig.pool.idle
13          },
14          query : {
15              raw : true
16          }
17      }
18  )
19
20  const profsModel = require('./profsModel')(sequelize,DataTypes);
21
22  sequelize.sync();
23
24  module.exports = {
25      db : sequelize,
26      profsModel : profsModel
27  }
```

LES MODELES

- Quelques petites remarques :
  - Un modèle a besoin de l'objet sequelize (celui créé avec le new ligne 4) et aussi des DataTypes pour être construit.
  - Cela limite les lieux dans le code où cela est possible.

- Le fichier `/models/index.js` est un bon endroit pour le faire.
- Chaque fois qu'on ajoutera un modèle à l'application, il faudra
  - Ajouter le `require` dans ce fichier
  - Ajouter l'export du modèle "construit"

MODIFICATION DU CONTRÔLEUR

- Dans notre contrôleur :
  - Ajouter le `require` du modèle
  - Modifier le code pour l'utiliser

/controllers/profsController.js

```
1  const {profsModel} = require('../models');
2
3  module.exports = {
4      getAll : function(req,res) {
5          profsModel.findAll()
6          .then((data) => {
7              res.status(200).json(
8                  {
9                      "status":'success',
10                     "data": data
11                 }
12             )
13         })
14     },
```

- La méthode doit être "pilotée par la promesse" retournée par le modèle.
- Ainsi la génération du résultat doit être à l'intérieur du `.then()`.

MODIFICATION DU CONTRÔLEUR

- Adapter toutes les autres méthodes sur ce mode.
- Supprimer le fichier `/data.js` et ses références.
- Tester.