

Introduction à Sequelize et ORM dans Node.js

Qu'est-ce qu'un ORM ?

En termes simples, un système ORM (Object-relational Mapper) est une technique dans laquelle vous utilisez un paradigme orienté objet pour créer un mappage entre l'application et la base de données afin d'effectuer directement la manipulation des données et la requête.

Lorsqu'il s'agit de récupérer et d'insérer des jointures et des relations, il suivra le même paradigme pour manipuler ou interroger les données liées aux opérations.

Qu'est-ce que Sequelize ?

Sequelize est un ORM TypeScript et Node.js moderne pour Oracle, Postgres, MySQL, MariaDB, SQLite et SQL Server, et plus encore. Avec une prise en charge solide des transactions, des relations, un chargement impatient et paresseux (eager and lazy loading), une réplication en lecture.

Ce tutoriel vous présentera Sequelize ORM et vous aidera à créer votre première application Sequelize NodeJS en utilisant l'approche *code-first*. **Il y a 4 questions/exercices dans ce document, assurez-vous d'y répondre.**

Code-first avec Sequelize

Créer un projet et se connecter à la base de données

Créez un nouveau projet NodeJS à l'aide de npm init dans un répertoire vide.

```
$ npm init
```

Ensuite, vous devrez installer les packages suivants :

```
$ npm install pg  
$ npm install sequelize  
$ npm install express body-parser
```

Pour utiliser Nodemon:

```
$ npm install --save-dev nodemon
```

Dans cette application, nous utiliserons PostgreSQL, si vous utilisez une autre base de données, vous devez installer manuellement le pilote de la base de données de votre choix :

L'un des éléments suivants :

```
$ npm install --save pg pg-hstore # Postgres
```

```
$ npm install --save mysql2
```

```
$ npm install --save mariadb
```

```
$ npm install --save sqlite3
```

```
$ npm install --save tedious # Microsoft SQL Server
```

```
$ npm install --save oracledb # Base de données Oracle
```

La première étape à faire est d'écrire du code pour connecter **Sequelize** à la base de données. Nous allons d'abord créer un fichier **index.js**, le point d'entrée de notre application. Créez ensuite un dossier "**database**" et un fichier **db.js** à l'intérieur de ce dossier.

```
$ touch index.js
$ mkdir database
$ touch database/db.js
```

Dans le fichier **db.js**, importez "**sequelize**", puis créez une nouvelle instance de sequelize en appelant **new Sequelize**. La fonction constructeur a besoin de certaines options telles que votre base de données, votre nom d'utilisateur, votre mot de passe, votre port et votre hôte. Assurez-vous de définir la configuration correcte.

```
const Sequelize = require("sequelize");

const sequelize = new Sequelize('votrebdd', 'votrelogin', 'votremotdepasse', {
  dialect: 'postgres',
  port: 5432,
  host: 'hôte'
});

module.exports = sequelize;
```

Définir un modèle

Les modèles sont l'essence de Sequelize. Un modèle est une abstraction qui représente une table dans votre base de données. Dans Sequelize, c'est une classe qui étend **Model**.

Le modèle indique à **Sequelize** plusieurs informations sur l'entité qu'il représente, telles que le nom de la table dans la base de données et les colonnes qu'elle contient (et leurs types de données).

Un modèle dans Sequelize a un nom. Ce nom ne doit pas nécessairement être le même que celui de la table qu'il représente dans la base de données. Habituellement, les modèles ont des noms au singulier (tels que **User**) tandis que les tables ont des noms au pluriel (tels que **Users**), bien que cela soit entièrement configurable.

Nous allons maintenant définir un modèle **Client** qui représente une table "**Client**" dans notre base de données. Créez un répertoire "**models**" et à l'intérieur, créez le fichier javascript **client.js**.

```
$ mkdir models
$ touch models/client.js
```

Les modèles peuvent être définis de deux manières équivalentes dans Sequelize. Une façon consiste à appeler **sequelize.define(modelName, attributs, options)**.

Pour apprendre avec un exemple, nous considérerons que nous voulons créer un modèle pour représenter les clients, qui ont un identifiant, un prénom et un nom. Nous voulons que notre modèle s'appelle **Client** et que la table qu'il représente s'appelle "**Client**" dans la base de données.

Cette façon de définir ce modèle est illustrée ci-dessous. Après avoir été défini, nous pouvons accéder à notre modèle avec **sequelize.models.Client**.

Le modèle **Client** aura les attributs suivants :

- id
 - Clé primaire
 - Type entier
 - non nul
 - devrait être unique
- firstname
 - Type String
 - non nul
- lastname
 - Type String
 - non nul

```

const sequelize = require("../database/db");
const { DataTypes } = require('sequelize');

const Client = sequelize.define('client', {
  id: {
    field: 'clientid',
    type: DataTypes.INTEGER,
    allowNull: false,
    unique: true,
    primaryKey: true
  },
  firstname: {
    field: 'firstName',
    type: DataTypes.STRING,
    allowNull: false
  },
  lastname: {
    field: 'lastName',
    type: DataTypes.STRING,
    allowNull: false
  }
});

module.exports = Client;

```

Synchronisation des définitions JS avec la base de données

Sequelize peut créer des tables pour vous et il vous suffit de dire à sequelize de le faire. Dans ce projet, nous le ferons dans le fichier **index.js**. Là-dedans, nous voulons nous assurer que tous nos modèles sont essentiellement transférés dans des tables ou obtenir une table qui leur appartient chaque fois que nous démarrons notre application. Et si la table existe déjà, sequelize ne la remplacera bien sûr pas par défaut bien que nous puissions lui dire de le faire.

Avant de synchroniser le modèle créé, nous allons créer un serveur express dans **index.js** et essayer de nous authentifier auprès de la base de données.



```
const express = require("express");
const bodyParser = require("body-parser");
const sequelize = require("../database/db");

const port = 3000;
const app = express();

async function authenticateDb(){
  try{
    await sequelize.authenticate();
    console.log("Connection successful!");
  }catch (error){
    console.log("Connection failed");
  }
}

app.use(bodyParser.json());

authenticateDb();

app.listen(port,()=>{
  console.log(`Le serveur ecoute sur le port ${port}`);
})
```

Si la configuration de votre base de données est correcte, lors de l'exécution du fichier index.js, vous devriez obtenir le message "Connection successful". En utilisant sequelize.authenticate, nous pouvons vérifier la connexion à notre base de données.

Lors de l'exécution de index.js, votre terminal doit ressembler à ceci :

```
Le serveur ecoute sur le port 3000
Executing (default): SELECT 1+1 AS result
Connection successful!
```

Lorsque vous définissez un modèle, vous indiquez à **Sequelize** quelques informations sur sa table dans la base de données. Cependant, que se passe-t-il si la table n'existe même pas dans la base de données ? Que se passe-t-il si elle existe, mais qu'elle a des colonnes différentes, moins de colonnes ou toute autre différence ?

C'est là qu'intervient la synchronisation des modèles. Un modèle peut être synchronisé avec la base de données en appelant **model.sync(options)**, une fonction asynchrone (qui renvoie une Promise). Avec cet appel, **Sequelize** effectuera automatiquement une requête SQL vers la base de données. Notez que cela ne modifie que la table dans la base de données, pas le modèle côté JavaScript.

- **Client.sync()** - Cela crée la table si elle n'existe pas (et ne fait rien si elle existe déjà)
- **Client.sync({ force: true })** - Cela crée la table, en la supprimant d'abord si elle existait déjà
- **Client.sync({ alter: true })** - Cela vérifie quel est l'état actuel de la table dans la base de données (quelles colonnes elle a, quels sont leurs types de données, etc.), puis effectue les modifications nécessaires dans la table pour faire il correspond au modèle.

Exemple:

```
const express = require("express");
const bodyParser = require("body-parser");
const sequelize = require("../database/db");
const Client = require("../models/client");

const port = 3000;
const app = express();

async function authenticateDb(){
  return sequelize.authenticate();
}

app.use(bodyParser.json());

authenticateDb()
  .then(()=>{
    console.log("Connection successful!");
    Client.sync();
  })
  .catch(()=>{
    console.log("Connection failed!");
  });

app.listen(port,()=>{
  console.log(`Le serveur ecoute sur le port ${port}`);
})
```

1 → Nous importons notre modèle

2 → Modifiez la fonction d'authentification pour renvoyer une promesse. Si la promesse est résolue, nous synchronisons le modèle. Si la promesse est rejetée, nous affichons un message d'erreur.

3 → Nous synchronisons le modèle. Si la table n'existe pas, sequelize la créera. Si la table existe, rien ne se passe.

4 → La connexion a échoué, nous affichons un message.

Lorsque vous exécutez `index.js`, vous devriez avoir une sortie dans votre terminal similaire à ceci :

```
Connection successful!
Le serveur ecoute sur le port 3000
Executing (default): SELECT 1+1 AS result
Executing (default): SELECT table_name FROM information_schema.tables WHERE table_schema = 'public' AND table_name = 'clients'
Executing (default): CREATE TABLE IF NOT EXISTS "clients" ("clientid" INTEGER NOT NULL UNIQUE , "firstName" VARCHAR(255) NOT NULL, "lastName" VARCHAR(255) NOT NULL, "createdAt" TIMESTAMP WITH TIME ZONE NOT NULL, "updatedAt" TIMESTAMP WITH TIME ZONE NOT NULL, PRIMARY KEY ("clientid"));
Executing (default): SELECT i.relname AS name, ix.indisprimary AS primary, ix.indisunique AS unique, ix.indkey AS indkey, array_agg(a.attnum) as column_indexes, array_agg(a.attname) AS column_names, pg_get_indexdef(ix.indexrelid) AS definition FROM pg_class t, pg_class i, pg_index ix, pg_attribute a WHERE t.oid = ix.indexrelid AND i.oid = ix.indexrelid AND a.attrelid = t.oid AND t.relkind = 'r' and t.relname = 'clients' GROUP BY i.relname, ix.indexrelid, ix.indisprimary, ix.indisunique, ix.indkey ORDER BY i.relname;
```

Que faire si vous avez plusieurs modèles et que vous souhaitez synchroniser tous les modèles en même temps ? Vous pouvez utiliser **`sequelize.sync()`** pour synchroniser automatiquement tous les modèles. Exemple:

```

const express = require("express");
const bodyParser = require("body-parser");
const sequelize = require("../database/db");
const Client = require("../models/client");

const port = 3000;
const app = express();

async function authenticateDb(){
  return sequelize.authenticate();
}

app.use(bodyParser.json());

authenticateDb()
  .then(async ()=>{
    console.log("Connection successful!");
    await sequelize.sync({ force: true }); // N'utilisez pas "force: true" en production
    console.log("All models were synchronized successfully.");
  })
  .catch(()=>{
    console.log("Connection failed!")
  });

app.listen(port, ()=>{
  console.log(`Le serveur ecoute sur le port ${port}`);
})

```

Les associations

Sequelize prend en charge les associations standard : One-To-One, One-To-Many et Many-To-Many.

Pour cela, Sequelize propose quatre types d'associations qu'il convient de combiner pour les créer :

- L'association HasOne
- L'association BelongsTo
- L'association HasMany
- L'association BelongsToMany

Association One-to-One

Supposons que chaque client a un passeport. Dans ce cas, la table clients dans la base de données serait la table parent et sa clé primaire apparaît comme clé étrangère dans la table enfant.

Dans le répertoire "models", créez un fichier passeport.js pour ajouter un nouveau modèle:

```
$ touch models/passport.js
```

```
const sequelize = require("../database/db");
const { DataTypes } = require('sequelize');

const Passport = sequelize.define('passport', {
  country: {
    type: DataTypes.STRING,
    allowNull: false
  },
  passportNumber: {
    type: DataTypes.INTEGER,
    allowNull: false,
    primaryKey: true
  },
  issueDate: {
    type: DataTypes.DATEONLY,
    allowNull: false
  },
  expirationDate: {
    type: DataTypes.DATEONLY,
    allowNull: false
  }
});
module.exports = Passport;
```

Créons un fichier **index.js** dans le dossier "**models**" pour importer toutes les entités de modèles.

```
1 // models/index.js
2 const Client = require("./client");
3 const Passport = require("./passport");
4
5
6 module.exports = {
7   Client : Client,
8   Passport : Passport
9 }
10
```

Nous pouvons maintenant ajouter les associations dans notre fichier index.js (le point d'entrée de l'application) comme suit :


```

const express = require("express");
const bodyParser = require("body-parser");
const sequelize = require("../database/db");
const {Client,Passport} = require("../models");

```

```

const port = 3000;
const app = express();

async function authenticateDb(){
  return sequelize.authenticate();
}

```

```
app.use(bodyParser.json());
```

```
Client.hasOne(Passport);
Passport.belongsTo(Client);
```

Le client a un (hasOne) passeport et le passeport appartient (belongsTo) à un client. hasOne est utilisé pour la table parent et belongsTo pour la table enfant.

```

authenticateDb()
  .then(async ()=>{
    console.log("Connection successful!")
    await sequelize.sync({ force: true }); // N'utilisez pas "force: true" en production
    console.log("All models were synchronized successfully.");
  })
  .catch(()=>{
    console.log("Connection failed!")
  });

app.listen(port,()=>{
  console.log(`Le serveur ecoute sur le port ${port}`);
})

```

Lors de l'exécution de "node index.js", vous pouvez vérifier la base de données et voir la table nouvellement créée "passports" qui contient la clé étrangère **clientId**.

```

postgres=# \d clients
                    Table "public.clients"
   Column   |          Type          | Collation | Nullable | Default
-----|-----|-----|-----|-----
 clientId | integer                |           | not null |
 firstName | character varying(255) |           | not null |
 lastName  | character varying(255) |           | not null |
 createdAt | timestamp with time zone |           | not null |
 updatedAt | timestamp with time zone |           | not null |
Indexes:
    "clients_pkey" PRIMARY KEY, btree (clientId)
Referenced by:
    TABLE "passports" CONSTRAINT "passports_clientId_fkey" FOREIGN KEY ("clientId") REFERENCES clients(clientId) ON UPDATE CASCADE ON DELETE SET NULL

```

```

postgres=# \d passports
                    Table "public.passports"
   Column   |          Type          | Collation | Nullable | Default
-----|-----|-----|-----|-----
 country   | character varying(255) |           | not null |
 passportNumber | integer                |           | not null |
 issueDate | date                   |           | not null |
 expirationDate | date                   |           | not null |
 createdAt | timestamp with time zone |           | not null |
 updatedAt | timestamp with time zone |           | not null |
 clientId | integer                |           |          |
Indexes:
    "passports_pkey" PRIMARY KEY, btree ("passportNumber")
Foreign-key constraints:
    "passports_clientId_fkey" FOREIGN KEY ("clientId") REFERENCES clients(clientId) ON UPDATE CASCADE ON DELETE SET NULL

```

Nous allons maintenant ajouter une fonction **seed_data()** qui va insérer deux clients et deux passeports dans la base de données. Ensuite, cette fonction liera chaque client à son passeport à l'aide d'une fonction **setter**. Lors de l'association d'un client à un passeport à l'aide d'une association one-to-one, Sequelize crée automatiquement des fonctions **getters** et **setters** telles que **setPassport** et **getPassport**. Notez que le nom de ces fonctions dépend du nom des modèles (singulier ou pluriel par exemple).

```
const express = require("express");
const bodyParser = require("body-parser");
const sequelize = require("../database/db");
const {Client,Passport} = require("../models");

const port = 3000;
const app = express();

async function authenticateDb(){
  return sequelize.authenticate();
}

async function seed_data(){
  let clients = [
    {id: 100, firstname: 'Abigail', lastname: 'Kylie'},
    {id: 110, firstname: 'Anna', lastname: 'Carolyn'}
  ]
  let passports = [
    {country: 'France', passportNumber: 111201, issueDate: '2014-12-07',
    expirationDate:'2024-12-07'},
    {country: 'USA', passportNumber: 901222, issueDate: '2019-11-07',
    expirationDate:'2027-11-07'}
  ]
  await Client.bulkCreate(clients);
  await Passport.bulkCreate(passports);

  for (let i = 0; i < passports.length; i++){
    passport = passports[i];
    cl = clients[i];
    pass = await Passport.findOne({where : { passportNumber: passport.passportNumber}});
    client = await Client.findOne({where : { id: cl.id}});
    await client.setPassport(pass);
  }
}

app.use(bodyParser.json());

Client.hasOne(Passport);
Passport.belongsTo(Client);

authenticateDb()
  .then(async ()=>{
    console.log("Connection successful!")
    await sequelize.sync({ force: true }); // N'utilisez pas "force: true" en production
    console.log("All models were synchronized successfully.");
    seed_data();
  })
  .catch(()=>{
    console.log("Connection failed!")
  });

app.listen(port,()=>{
  console.log(`Le serveur ecoute sur le port ${port}`);
});
```

```

async function seed_data(){
  let clients = [
    {id: 100, firstname: 'Abigail', lastname: 'Kylie'},
    {id: 110, firstname: 'Anna', lastname: 'Carolyn'}
  ]
  let passports = [
    {country: 'France', passportNumber: 111201, issueDate: '2014-12-07',
    expirationDate: '2024-12-07'},
    {country: 'USA', passportNumber: 901222, issueDate: '2019-11-07',
    expirationDate: '2027-11-07'}
  ]
  await Client.bulkCreate(clients); // 1
  await Passport.bulkCreate(passports); // 2
  for (let i = 0; i < passports.length; i++){
    passport = passports[i];
    cl = clients[i];
    pass = await Passport.findOne({where : { passportNumber: passport.passportNumber}}); // 3
    client = await Client.findOne({where : { id: cl.id}}); // 4
    await client.setPassport(pass); // 5
  }
}

```

1 INSERT INTO "clients" ("clientId","firstName","lastName")
VALUES (100,'Abigail','Kylie'),(110,'Anna','Carolyn')

2 INSERT INTO "passports" ("country","passportNumber","issueDate","expirationDate")
VALUES ('France',111201,'2014-12-07','2024-12-07'),('USA',901222,'2019-11-07','2027-11-07')

3 Select * from passports where passportNumber = 111201

4 Select * from clients where clientId = 100

5 UPDATE "passports" SET "clientId"=100 WHERE "passportNumber" = 111201

Copiez et collez ce code :

```

async function seed_data(){
  let clients = [
    {id: 100, firstname: 'Abigail', lastname: 'Kylie'},
    {id: 110, firstname: 'Anna', lastname: 'Carolyn'}
  ]
  let passports = [
    {country: 'France', passportNumber: 111201, issueDate:
'2014-12-07', expirationDate: '2024-12-07'},
    {country: 'USA', passportNumber: 901222, issueDate:
'2019-11-07', expirationDate: '2027-11-07'}
  ]
  await Client.bulkCreate(clients);
  await Passport.bulkCreate(passports);

  for (let i = 0; i < passports.length; i++){
    passport = passports[i];
    cl = clients[i];
    pass = await Passport.findOne({where : { passportNumber:
passport.passportNumber}});
    client = await Client.findOne({where : { id: cl.id}});
    await client.setPassport(pass);
  }
}

```

Vous pouvez vérifier les deux tables après avoir exécuté index.js :

```

|postgres=# select * from clients;
|-----+-----+-----+-----+-----+
|clientId | firstName | lastName | createdAt | updatedAt |
|-----+-----+-----+-----+-----+
| 100     | Abigail  | Kylie   | 2022-11-22 00:00:41.154+01 | 2022-11-22 00:00:41.154+01 |
| 110     | Anna     | Carolyn | 2022-11-22 00:00:41.154+01 | 2022-11-22 00:00:41.154+01 |
|(2 rows)

```

```

|postgres=# select * from passports;
|-----+-----+-----+-----+-----+-----+-----+
|country | passportNumber | issueDate | expirationDate | createdAt | updatedAt | clientId |
|-----+-----+-----+-----+-----+-----+-----+
| France | 111201         | 2014-12-07 | 2024-12-07     | 2022-11-22 00:00:41.161+01 | 2022-11-22 00:00:41.175+01 | 100     |
| USA    | 901222         | 2019-11-07 | 2027-11-07     | 2022-11-22 00:00:41.161+01 | 2022-11-22 00:00:41.181+01 | 110     |
|(2 rows)

```

Pour rendre les choses plus excitantes, créons une route dans notre fichier **index.js** permettant à l'utilisateur de publier des données relatives aux utilisateurs et aux passeports (**HTTP POST**). Dans ce projet, par souci de démonstration, nous n'utiliserons pas le modèle **Routeurs-Contrôleurs-Modèles/Services**. Nous mettrons toutes les routes dans notre fichier **index.js** (ce qui n'est pas recommandé lorsque l'on travaille sur une vraie application).

1- Ajoutez une route `/clients` qui accepte les messages HTTP POST. En utilisant la fonction **.create**, nous allons créer un nouveau client. Sequelize a créé la fonction **.createPassport** lorsque nous avons fait les associations afin que nous puissions l'utiliser pour créer un passeport lié à un certain client.

```
app.post("/clients", async (req, res) => {
  let data = {
    id: req.body.id,
    firstname: req.body.firstname,
    lastname: req.body.lastname,
    passport_country: req.body.passport_country,
    passportNumber: req.body.passportNumber,
    passport_issue_date: req.body.passport_issue_date,
    passport_expiry_date: req.body.passport_expiry_date
  }
  let newClient = await Client.create({id: data.id, firstname: data.firstname, lastname: data.lastname});
  await newClient.createPassport({
    country: data.passport_country, passportNumber: data.passportNumber,
    issueDate: data.passport_issue_date, expirationDate: data.passport_expiry_date,
  });
  return res.status(200).send({status: 1, data: "Data inserted successfully!!!"});
});
```

2- Testez votre API avec Postman ou cURL :

The screenshot shows the Postman interface for a POST request to `localhost:3000/clients`. The request body is a JSON object with the following structure:

```
1 {
2   "id": 120,
3   "firstname": "Joshua",
4   "lastname": "Fraser",
5   "passport_country": "USA",
6   "passportNumber": 901105,
7   "passport_issue_date": "2021-05-06",
8   "passport_expiry_date": "2030-05-06"
9 }
```

The response is a JSON object with the following structure:

```
1 {
2   "status": 1,
3   "data": "Data inserted successfully!!!"
4 }
```

The status bar at the bottom indicates a `200 OK` response with a time of `61 ms`.

3- Vérifiez les données de votre base de données :

```
|postgres=# select * from clients;
 clientid | firstName | lastName |          createdAt          |          updatedAt
-----|-----|-----|-----|-----
    100 | Abigail   | Kylie   | 2022-11-22 00:34:18.004+01 | 2022-11-22 00:34:18.004+01
    110 | Anna     | Carolyn | 2022-11-22 00:34:18.004+01 | 2022-11-22 00:34:18.004+01
    120 | Joshua   | Fraser  | 2022-11-22 00:37:22.866+01 | 2022-11-22 00:37:22.866+01
(3 rows)
```

```
|postgres=# select * from passports;
 country | passportNumber | issueDate | expirationDate |          createdAt          |          updatedAt          | clientid
-----|-----|-----|-----|-----|-----|-----
 France |      111201    | 2014-12-07 | 2024-12-07    | 2022-11-22 00:34:18.012+01 | 2022-11-22 00:34:18.027+01 |      100
 USA    |      901222    | 2019-11-07 | 2027-11-07    | 2022-11-22 00:34:18.012+01 | 2022-11-22 00:34:18.031+01 |      110
 USA    |      901105    | 2021-05-06 | 2030-05-06    | 2022-11-22 00:37:22.882+01 | 2022-11-22 00:37:22.882+01 |      120
(3 rows)
```

Association One-to-Many

Prenons le cas où les articles sont vendus. Nous pouvons immédiatement identifier deux entités : SALE et ITEM. Une vente (SALE) peut contenir de nombreux articles (ITEM) et un article peut apparaître dans de nombreuses ventes. Chaque vente est réalisée par un client, et un client peut réaliser plusieurs ventes. Intéressons-nous maintenant à la relation entre un client et une vente.

Les associations One-To-Many connectent une source à plusieurs cibles, alors que toutes ces cibles ne sont connectées qu'à cette seule source. Cela signifie que, contrairement à l'association One-To-One, dans laquelle nous devons choisir où placer la clé étrangère, il n'y a qu'une seule option dans les associations One-To-Many. Par exemple, si un client a plusieurs ventes (et de cette façon chaque vente appartient à un client), alors la seule implémentation sensée est d'avoir une colonne clientId dans la table SALES. L'inverse est impossible, puisqu'un Client a plusieurs Ventes.

Dans cet exemple, nous avons les modèles **Client** et **Sale**. Nous voulons dire à Sequelize qu'il existe une relation un à plusieurs entre eux, ce qui signifie qu'un **Client** a plusieurs **Sales**, tandis que chaque **Sale** appartient à un seul **Client**.

Ajoutons un nouveau modèle "sale.js" dans notre répertoire "models" :

```
$ touch models/sale.js
```

```
// models/sale.js
const sequelize = require("../database/db");
const { DataTypes } = require('sequelize');

const Sale = sequelize.define('sale',{
  saleno: {
    field: 'saleno',
    type: DataTypes.INTEGER,
    allowNull: false,
    unique: true,
    primaryKey: true,
    autoIncrement: true
  },
  saledate: {
    field: 'saledate',
    type: DataTypes.DATEONLY,
    allowNull: false
  },
  saletext: {
    field: 'saletext',
    type: DataTypes.STRING,
    allowNull: false
  }
});

module.exports = Sale;
```

```
// models/index.js
const Client = require("./client");
const Passport = require("./passport");
const Sale = require("./sale")

module.exports = {
  Client : Client,
  Passport : Passport,
  Sale: Sale
}
```

Dans **index.js** (point d'entrée de l'application), assurez-vous d'importer également "**Sale**" en haut du fichier :

```
const {Client, Passport, Sale} = require("./models");
```

Ajoutez les associations suivantes :

```
Client.hasOne(Passport);
Passport.belongsTo(Client);

Sale.belongsTo(Client, {options: {constraints:true, onDelete: 'CASCADE'}});
Client.hasMany(Sale);

authenticateDb() Promise<any>
  .then(async ()=>{
    console.log("Connection successful!")
    await sequelize.sync({ force: true }); // N'utilisez pas "force: true" en production
    console.log("All models were synchronized successfully.");
    seed_data();
  }) Promise<any>
  .catch(()=>{
    console.log("Connection failed!")
  });

app.listen(port, ()=>{
  console.log(`Le serveur écoute sur le port ${port}`);
});
```

ajouter les contraintes ON DELETE et ON UPDATE

Exécutez votre fichier **index.js** et vérifiez la table dans votre base de données :

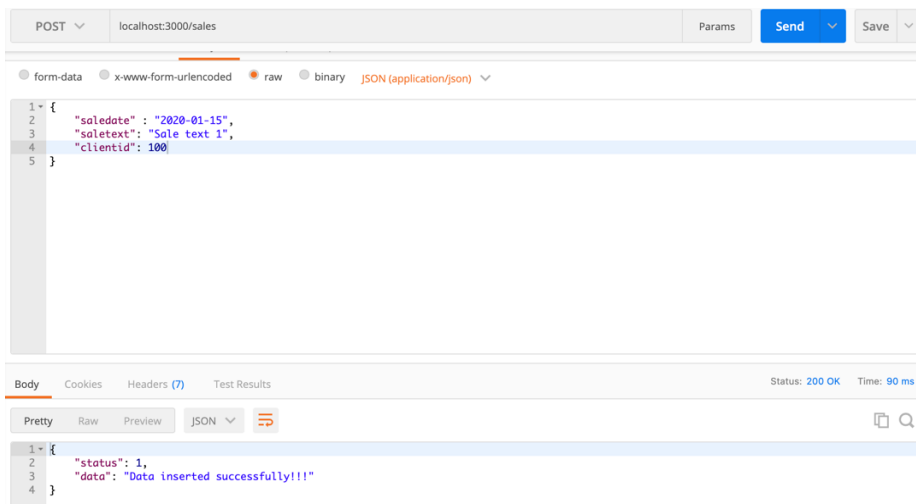
```
postgres=# \d sales;
          Table "public.sales"
   Column |          Type          | Collation | Nullable | Default
-----|-----|-----|-----|-----
 saleno   | integer                |           | not null | nextval('sales_saleno_seq'::regclass)
 saledate | date                   |           | not null |
 saletext | character varying(255) |           | not null |
 createdAt | timestamp with time zone |           | not null |
 updatedAt | timestamp with time zone |           | not null |
 clientId | integer                 |           | not null |
Indexes:
    "sales_pkey" PRIMARY KEY, btree (saleno)
Foreign-key constraints:
    "sales_clientId_fkey" FOREIGN KEY ("clientId") REFERENCES clients(clientid) ON UPDATE CASCADE ON DELETE CASCADE
```

Notez que, dans les deux modèles ci-dessus (**Client** et **Sale**), le nom de la table (**clients/sales**) n'a jamais été explicitement défini. Cependant, le nom du modèle a été donné (**client/sale**). Par défaut, lorsque le nom de la table n'est pas donné, **Sequelize** met automatiquement au pluriel le nom du modèle et l'utilise comme nom de table.

Exercice 1 :

Créez une route **POST /sales** dans laquelle vous pouvez insérer une nouvelle vente dans la base de données avec une requête HTTP Post. Remplissez le tableau des ventes (sales) avec des données factices (au moins 2 ventes pour chaque utilisateur).

Exemple requête HTTP Post :



Exemple de contenu de table :

```
[postgres=# select * from sales;
 saleno | saledate | saletext |          createdAt          |          updatedAt          | clientid
-----+-----+-----+-----+-----+-----
      1 | 2020-01-25 | Sale text 4 | 2022-11-23 19:37:28.147+01 | 2022-11-23 19:37:28.147+01 |      110
      2 | 2020-01-25 | Sale text 3 | 2022-11-23 19:37:32.882+01 | 2022-11-23 19:37:32.882+01 |      110
      3 | 2020-01-21 | Sale text 2 | 2022-11-23 19:37:39.81+01 | 2022-11-23 19:37:39.81+01 |      100
      4 | 2020-01-21 | Sale text 1 | 2022-11-23 19:37:41.948+01 | 2022-11-23 19:37:41.948+01 |      100
(4 rows)
```

Chargement impatient vs chargement paresseux (Eager Loading vs Lazy Loading)

Les concepts de **Eager Loading** et de **Lazy Loading** sont fondamentaux pour comprendre le fonctionnement des associations de récupération dans **Sequelize**. Le Lazy Loading fait référence à la technique consistant à récupérer les données associées uniquement lorsque vous le souhaitez vraiment ; **Eager Loading**, quant à lui, fait référence à la technique consistant à tout récupérer en même temps, depuis le début, avec une requête plus large.

Testons d'abord le **Lazy Loading**. Créez une route **GET /clients/lazy/:id** pour obtenir toutes les informations relatives à un client avec un identifiant donné. Notez que dans l'exemple ci-dessous, nous avons fait deux requêtes SQL, ne récupérant les ventes associées que lorsque nous voulions les utiliser. Cela peut être particulièrement utile si nous pouvons ou non avoir besoin des ventes, peut-être voulons-nous les récupérer sous condition, seulement dans quelques cas ; de cette façon, nous pouvons gagner du temps et de la mémoire en ne le récupérant que lorsque cela est nécessaire.


```

app.get("/clients/lazy/:id", async (req,res)=>{
  let id = req.params.id;
  // rechercher par clé primaire
  const client = await Client.findByPk(id);
  let response = {
    client_id: client.clientid,
    client_fullname: client.firstname + " " + client.lastname
  };
  // obtenir toutes les ventes liées à ce client
  const sales = await client.getSales();
  response.sales = sales;
  return res.status(200).send({status: 1, data: response});
})

```

The screenshot shows a REST client interface with the following details:

- Method:** GET
- URL:** localhost:3000/clients/lazy/100
- Status:** 200 OK
- Time:** 21 ms
- Response Body (JSON):**

```

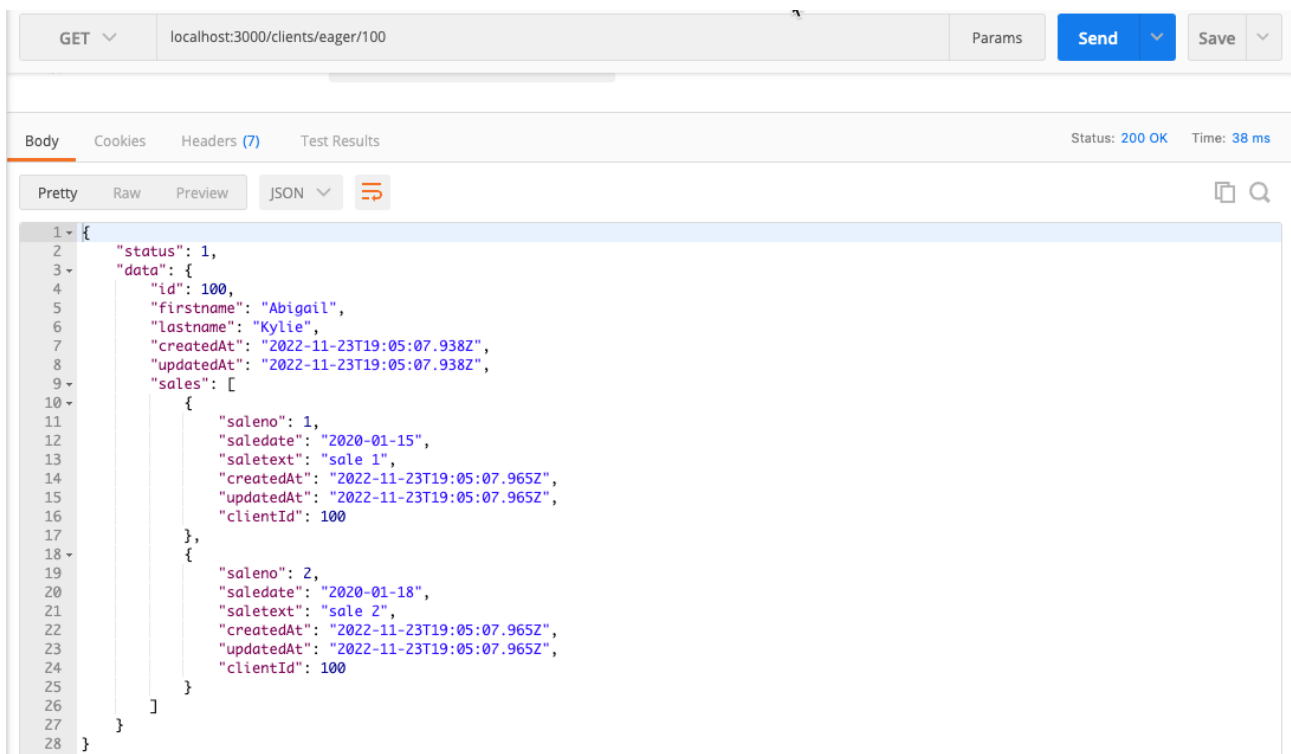
{
  "status": 1,
  "data": {
    "client_fullname": "Abigail Kylie",
    "sales": [
      {
        "saleno": 1,
        "saledate": "2020-01-15",
        "saletext": "sale 1",
        "createdAt": "2022-11-23T18:57:04.857Z",
        "updatedAt": "2022-11-23T18:57:04.857Z",
        "clientId": 100
      },
      {
        "saleno": 2,
        "saledate": "2020-01-18",
        "saletext": "sale 2",
        "createdAt": "2022-11-23T18:57:04.857Z",
        "updatedAt": "2022-11-23T18:57:04.857Z",
        "clientId": 100
      }
    ]
  }
}

```

Testons maintenant le **Eager Loading**. Créez une route **GET /clients/eager/:id** pour obtenir toutes les informations relatives à un client avec un identifiant donné. **Eager Loading** est effectué dans Sequelize à l'aide de l'option **include**. Observez dans

l'exemple ci-dessous qu'une seule requête a été effectuée sur la base de données (qui apporte les données associées avec l'instance).

```
app.get("/clients/eager/:id", async (req,res)=>{
  let id = req.params.id;
  // rechercher par clé primaire
  const client = await Client.findByPk(id,{include: Sale});
  return res.status(200).send({status: 1, data: client});
})
```



The screenshot shows a REST client interface with a GET request to `localhost:3000/clients/eager/100`. The response is a JSON object with the following structure:

```
{
  "status": 1,
  "data": {
    "id": 100,
    "firstname": "Abigail",
    "lastname": "Kylie",
    "createdAt": "2022-11-23T19:05:07.938Z",
    "updatedAt": "2022-11-23T19:05:07.938Z",
    "sales": [
      {
        "saleno": 1,
        "saledate": "2020-01-15",
        "saletext": "sale 1",
        "createdAt": "2022-11-23T19:05:07.965Z",
        "updatedAt": "2022-11-23T19:05:07.965Z",
        "clientId": 100
      },
      {
        "saleno": 2,
        "saledate": "2020-01-18",
        "saletext": "sale 2",
        "createdAt": "2022-11-23T19:05:07.965Z",
        "updatedAt": "2022-11-23T19:05:07.965Z",
        "clientId": 100
      }
    ]
  }
}
```

Exercice 2 :

Créez une route **PUT /clients/:id** qui prend l'id du client dans les paramètres et ses prénom et nom dans le corps (**body**) et mettez à jour la table **clients**. Pour cette requête, vous utiliserez la méthode **update** et la clause **where** avec sequelize.

Voici la syntaxe de mise à jour d'une requête extraite de la documentation de sequelize :

Simple UPDATE queries

Update queries also accept the `where` option, just like the read queries shown above.

```
// Change everyone without a last name to "Doe"
await User.update({ lastName: "Doe" }, {
  where: {
    lastName: null
  }
});
```

Vous devriez avoir des résultats similaires :

The screenshot shows a REST client interface. The top bar indicates a PUT request to localhost:3000/clients/100. The 'Body' tab is selected, showing a JSON payload: `{ "firstname": "Elon", "lastname": "Musk" }`. The response status is 200 OK, and the response body is `"Client updated successfully!!"`.

```
postgres=# select * from clients;
 clientid | firstName | lastName |          createdAt          |          updatedAt          |
-----+-----+-----+-----+-----+
      110 | Anna     | Carolyn | 2022-11-23 20:17:10.075+01 | 2022-11-23 20:17:10.075+01 |
      100 | Elon    | Musk    | 2022-11-23 20:17:10.075+01 | 2022-11-23 20:17:13.235+01 |
(2 rows)
```

Exercice 3 :

créer une route **DELETE /sales/:id** qui prend l'id d'une vente dans les paramètres et supprimer la vente de la table **"sales"**. Pour cette requête, vous utilisez la méthode **"destroy"** et la clause **where** avec **Sequelize**.

Voici la syntaxe d'une opération de suppression tirée de la documentation sequelize:

Simple DELETE queries

Delete queries also accept the `where` option, just like the read queries shown above.

```
// Delete everyone named "Jane"
await User.destroy({
  where: {
    firstName: "Jane"
  }
});
```

Vous devriez avoir des résultats similaires :

DELETE localhost:3000/sales/1

Authorization Headers (1) Body Pre-request Script Tests

Type No Auth

Body Cookies Headers (7) Test Results Status: 200 OK Time: 53 ms

Pretty Raw Preview JSON

```
1 {
2   "status": 1,
3   "data": "Sale deleted successfully!!!"
4 }
```

```
postgres=# select * from sales;
 saleno | saledate | saletext |          createdAt          |          updatedAt          | clientId
-----|-----|-----|-----|-----|-----
      2 | 2020-01-18 | sale 2 | 2022-11-23 20:40:30.306+01 | 2022-11-23 20:40:30.306+01 |      100
      3 | 2020-01-21 | sale 3 | 2022-11-23 20:40:30.306+01 | 2022-11-23 20:40:30.306+01 |      110
      4 | 2020-01-25 | sale 4 | 2022-11-23 20:40:30.306+01 | 2022-11-23 20:40:30.306+01 |      110
(3 rows)
```

Faisons une requête intéressante avec Sequelize. Créez une route **GET /sales/topclient**. Nous voulons renvoyer le client qui a les ventes les plus élevées et dont les ventes sont supérieures à 2.

Supposons que dans votre table des ventes (**sales**), vous ayez ces enregistrements ou quelque chose de similaire (un client qui a plus de 2 ventes):

```
postgres=# select * from sales;
 saleno | saledate | saletext |          createdAt          |          updatedAt          | clientId
-----|-----|-----|-----|-----|-----
      1 | 2020-01-15 | sale 1 | 2022-11-23 23:49:22.194+01 | 2022-11-23 23:49:22.194+01 |      100
      2 | 2020-01-18 | sale 2 | 2022-11-23 23:49:22.194+01 | 2022-11-23 23:49:22.194+01 |      100
      3 | 2020-01-21 | sale 3 | 2022-11-23 23:49:22.194+01 | 2022-11-23 23:49:22.194+01 |      110
      4 | 2020-01-25 | sale 4 | 2022-11-23 23:49:22.194+01 | 2022-11-23 23:49:22.194+01 |      110
      5 | 2020-02-25 | sale 5 | 2022-11-23 23:49:22.194+01 | 2022-11-23 23:49:22.194+01 |      110
```

Vous souhaitez recréer la requête suivante dans sequelize :

Query Query History

```

1 select concat_ws(' ', clients."firstName", clients."lastName") as "fullName"
2 count(*) as "numsales"
3 from clients
4 JOIN sales
5 on clients."clientId" = sales."clientId"
6 group by concat_ws(' ', clients."firstName", clients."lastName")
7 having count(*) > 1
8 order by "numsales" desc
9 limit 1;
10
11
12

```

Data output Messages Notifications

| | fullName | numsales |
|---|--------------|----------|
| | text | bigint |
| 1 | Anna Carolyn | 3 |

Il n'est pas toujours possible pour **Sequelize** de prendre en charge toutes les fonctionnalités SQL de manière propre. La requête ci-dessus n'est certainement pas simple. Il peut parfois être préférable d'écrire la requête SQL vous-même.

Ceci peut être fait de deux façons:

- Soit écrire vous-même une requête brute complète,
- ou utilisez la fonction **literal()** fournie par **Sequelize** pour insérer du SQL brut presque n'importe où dans les requêtes construites par **Sequelize**.

La requête ci-dessus peut être effectuée comme suit dans **Sequelize** :

```

app.get("/sales/topclient", async (req,res)=>{
  try{
    const client = await Client.findAll({
      attributes: ['id',
        [sequelize.literal('concat_ws(\ ' \ ', "firstName", "lastName)'), 'fullName'],
        [sequelize.fn('COUNT', 'clientId'), 'numsales']],
      include: [{model: Sale,attributes: []}],
      group: ['id',sequelize.literal('concat_ws(\ ' \ ', "firstName", "lastName")']],
      having: sequelize.where(sequelize.fn('COUNT', sequelize.col('"clientId"')), '>=', 2),
      order: [[sequelize.fn('COUNT', sequelize.col('"clientId"')), 'DESC']]
    });
    return res.status(200).send({status: 1, data: client[0]});
  }catch (e){
    console.error(e);
  }
});

```

The screenshot shows a REST client interface. At the top, the method is 'GET' and the URL is 'localhost:3000/sales/topclient'. Below the URL bar, there are tabs for 'Authorization', 'Headers (1)', 'Body', 'Pre-request Script', and 'Tests'. The 'Type' dropdown is set to 'No Auth'. Below this, there are tabs for 'Body', 'Cookies', 'Headers (7)', and 'Test Results'. The 'Body' tab is selected, and the response is displayed in 'Pretty' format as JSON. The JSON response is:

```
{
  "status": 1,
  "data": {
    "id": 110,
    "fullName": "Anna Carolyn",
    "numsales": "3"
  }
}
```

Association Many-to-Many

Dans votre répertoire "**models**", créez un nouveau modèle javascript "**item.js**". Chaque article aura un numéro d'article, un nom d'article, un type et une couleur.

```
$ touch models/item.js
```

Vous pouvez arrêter l'auto-pluralisation effectuée par **Sequelize** à l'aide de l'option **freezeTableName: true**. De cette façon, Sequelize déduira que le nom de la table est égal au nom du modèle, sans aucune modification.

Par défaut, **Sequelize** ajoute automatiquement les champs **createdAt** et **updatedAt** à chaque modèle, en utilisant le type de données **DataTypes.DATE**. Ces champs sont également gérés automatiquement - chaque fois que vous utilisez **Sequelize** pour créer ou mettre à jour quelque chose, ces champs seront définis correctement. Le champ **createdAt** contiendra l'horodatage représentant le moment de la création, et **updatedAt** contiendra l'horodatage de la dernière mise à jour. Ce comportement peut être désactivé pour un modèle avec l'option **timestamps : false**.

```

const sequelize = require("../database/db");
const { DataTypes } = require('sequelize');

const Item = sequelize.define('item',{
  itemno: {
    field: 'itemno',
    type: DataTypes.INTEGER,
    allowNull: false,
    unique: true,
    primaryKey: true,
    autoIncrement: true
  },
  itemname: {
    field: 'itemname',
    type: DataTypes.STRING,
    allowNull: false
  },
  itemtype: {
    field: 'itemtype',
    type: DataTypes.STRING,
    allowNull: false
  },
  itemcolor: {
    field: 'itemcolor',
    type: DataTypes.STRING,
    allowNull: false
  }
},{
  timestamps: false,
  freezeTableName : true
});

module.exports = Item;

```

Lorsque nous avons une relation **m:m**, nous créons une entité associative pour stocker des données sur la relation. Dans ce cas, nous devons stocker des données sur les articles vendus. Nous ne pouvons pas stocker les données avec **SALE** car une vente peut avoir de nombreux articles et une instance d'une entité ne stocke que des faits à valeur unique. De même, nous ne pouvons pas stocker de données avec **ITEM** car un article peut apparaître dans de nombreuses ventes. Comme nous ne pouvons pas stocker de données dans **SALE** ou **ITEM**, nous devons créer une autre entité pour stocker des données sur la relation **m:m**. Pour cela, dans votre répertoire "**models**", créez le modèle javascript "**saleitem.js**".

```
$ touch models/saleitem.js
```

```

const sequelize = require("../database/db");
const { DataTypes } = require('sequelize');
const SaleItem = sequelize.define('saleItem',{
  id: {
    field: 'id',
    type: DataTypes.INTEGER,
    allowNull: false,
    unique: true,
    primaryKey: true,
    autoIncrement: true
  },
  quantity: {
    field: 'quantity',
    type: DataTypes.INTEGER,
    allowNull: false
  },
  price: {
    field: 'price',
    type: DataTypes.FLOAT,
    allowNull: false
  }
}, {
  freezeTableName: true,
  timestamps: false
});
module.exports = SaleItem;

```

Mettez à jour le fichier **models/index.js** comme suit :

```

// models/index.js
const Client = require("./client");
const Passport = require("./passport");
const Sale = require("./sale");
const Item = require("./item");
const SaleItem = require("./saleitem");

module.exports = {
  Client : Client,
  Passport : Passport,
  Sale: Sale,
  Item: Item,
  SaleItem: SaleItem
}

```


Dans **index.js** (point d'entrée de l'application), assurez-vous d'importer également **"Item"** et **"SaleItem"** en haut du fichier :

```
const {Client, Passport, Sale, Item, SaleItem} = require("../models");
```

Pour cet exemple, nous allons considérer les modèles **Sale** et **Item**. Une vente peut contenir de nombreux articles et un article peut être impliqué dans de nombreuses ventes. La table de jonction qui gardera une trace des associations s'appellera **SaleItem**, qui contiendra les clés étrangères **saleno** et **itemno**.

La principale façon de procéder dans **Sequelize** est la suivante :

```
//One-to-One
Client.hasOne(Passport);
Passport.belongsTo(Client);

//One-to-Many
Sale.belongsTo(Client, options: {constraints:true, onDelete: 'CASCADE'});
Client.hasMany(Sale);

//Many-to-Many
Sale.belongsToMany(Item, options: {through: SaleItem, foreignKey: 'saleno'});
Item.belongsToMany(Sale, options: {through: SaleItem, foreignKey: 'itemno'});
```

Exécutez **"node index.js"** et vérifiez votre base de données.

```
postgres=# \d "item"
          Table "public.item"
   Column |          Type          | Collation | Nullable |          Default
-----|-----|-----|-----|-----
 itemno  | integer                |           | not null | nextval('item_itemno_seq'::regclass)
 itemname| character varying(255)|           | not null |
 itemtype| character varying(255)|           | not null |
 itemcolor| character varying(255)|           | not null |
Indexes:
 "item_pkey" PRIMARY KEY, btree (itemno)
Referenced by:
 TABLE "saleItem" CONSTRAINT "saleItem_itemno_fkey" FOREIGN KEY (itemno) REFERENCES item(itemno) ON UPDATE CASCADE ON DELETE CASCADE

postgres=# \d "saleItem"
          Table "public.saleItem"
   Column |          Type          | Collation | Nullable |          Default
-----|-----|-----|-----|-----
 id       | integer                |           | not null | nextval('"saleItem_id_seq"'::regclass)
 quantity| integer                |           | not null |
 price    | double precision       |           | not null |
 saleno   | integer                |           |           |
 itemno   | integer                |           |           |
Indexes:
 "saleItem_pkey" PRIMARY KEY, btree (id)
 "saleItem_saleno_itemno_key" UNIQUE CONSTRAINT, btree (saleno, itemno)
Foreign-key constraints:
 "saleItem_itemno_fkey" FOREIGN KEY (itemno) REFERENCES item(itemno) ON UPDATE CASCADE ON DELETE CASCADE
 "saleItem_saleno_fkey" FOREIGN KEY (saleno) REFERENCES sales(saleno) ON UPDATE CASCADE ON DELETE CASCADE
```

Mettez à jour la fonction `seed_data()` pour ajouter des données factices aux tables nouvellement créées.

```
async function seed_data(){
  let clients = [
    {id: 100, firstname: 'Abigail', lastname: 'Kylie'},
    {id: 110, firstname: 'Anna', lastname: 'Carolyn'}
  ]
  let passports = [
    {country: 'France', passportNumber: 111201, issueDate:
'2014-12-07', expirationDate: '2024-12-07'},
    {country: 'USA', passportNumber: 901222, issueDate:
'2019-11-07', expirationDate: '2027-11-07'}
  ]
  await Client.bulkCreate(clients);
  await Passport.bulkCreate(passports);

  for (let i = 0; i < passports.length; i++){
    passport = passports[i];
    cl = clients[i];
    pass = await Passport.findOne({where : { passportNumber:
passport.passportNumber}});
    client = await Client.findOne({where : { id: cl.id}});
    await client.setPassport(pass);
  }

  // insert dummy sales
  let dummy_sales = [
    {saleDate: '2020-01-15', saleText: 'sale 1', clientId: 100},
    {saleDate: '2020-01-18', saleText: 'sale 2', clientId: 100},
    {saleDate: '2020-01-21', saleText: 'sale 3', clientId: 110},
    {saleDate: '2020-01-25', saleText: 'sale 4', clientId: 110},
    {saleDate: '2020-02-25', saleText: 'sale 5', clientId: 110},
  ]
  await Sale.bulkCreate(dummy_sales);

  // insert dummy items
  let dummy_items = [
    {itemName: 'Pocket knife-Nile', itemType: "E", itemcolor: "Brown"},
    {itemName: 'Pocket knife-Avon', itemType: "E", itemcolor: "Brown"},
    {itemName: 'Compass', itemType: "N", itemcolor: "_"},
    {itemName: 'Hammock', itemType: "F", itemcolor: "Khaki"},
    {itemName: 'Safari cooking kit', itemType: "E", itemcolor: "_"}
  ]
  await Item.bulkCreate(dummy_items);

  let dummy_sale_items = [
    {quantity: 2, price: 10, saleno: 1, itemno: 3},
    {quantity: 3, price: 24, saleno: 1, itemno: 1},
    {quantity: 1, price: 8, saleno: 2, itemno: 1},
    {quantity: 2, price: 10, saleno: 3, itemno: 3},
    {quantity: 3, price: 60, saleno: 3, itemno: 5},
    {quantity: 1, price: 20, saleno: 4, itemno: 4},
    {quantity: 1, price: 10, saleno: 4, itemno: 2},
    {quantity: 2, price: 10, saleno: 5, itemno: 3}
  ]

  await SaleItem.bulkCreate(dummy_sale_items);
}
```

```

[postgres=# select * from "item";
 itemno |      itemname      | itemtype | itemcolor
-----+-----+-----+-----
      1 | Pocket knife-Nile | E        | Brown
      2 | Pocket knife-Avon | E        | Brown
      3 | Compass            | N        | -
      4 | Hammock            | F        | Khaki
      5 | Safari cooking kit | E        | -
(5 rows)

```

```

[postgres=# select * from "saleItem";
 id | quantity | price | saleno | itemno
----+-----+-----+-----+-----
  1 |         2 |    10 |       1 |      3
  2 |         3 |    24 |       1 |      1
  3 |         1 |     8 |       2 |      1
  4 |         2 |    10 |       3 |      3
  5 |         3 |    60 |       3 |      5
  6 |         1 |    20 |       4 |      4
  7 |         1 |    10 |       4 |      2
  8 |         2 |    10 |       5 |      3
(8 rows)

```

Défi : créez une route HTTP GET /items/lowestup qui envoie l'article au prix le plus bas.

Il existe une fonctionnalité intéressante dans Sequelize appelée **Virtual fields**. Les champs virtuels sont des champs que Sequelize remplit, mais en réalité ils n'existent même pas dans la base de données. Par exemple, disons que nous avons les attributs de prix et de quantité pour un **SaleItem**. Ce serait bien d'avoir un moyen simple d'obtenir directement le prix unitaire (**unitprice**) ! Nous pouvons combiner l'idée des getters avec le type de données spécial que Sequelize fournit pour ce genre de situation :

DataTypes.VIRTUAL.

Le prix unitaire d'un article peut être calculé en divisant le prix par la quantité dans la table "**saleItem**". En SQL, nous pouvons simplement le faire en ajoutant une colonne calculée et en lui donnant un alias. Dans **Sequelize**, pour ce faire, ouvrez votre modèle **SaleItem** (**models/saleitem.js**) et ajoutez un attribut "**unitprice**" au modèle et dans la fonction **getter**, vous pouvez renvoyer le résultat du **prix/quantité** comme suit. Le champ **VIRTUAL** n'entraîne pas l'existence d'une colonne dans la table. En d'autres termes, le modèle ci-dessous n'aura pas de colonne de prix unitaire. Cependant, il semblera l'avoir! Notez que pour pouvoir obtenir le prix unitaire à partir de la fonction **findAll()**, vous devez vous assurer que les colonnes utilisées pour obtenir ce champ (dans ce cas la quantité et le prix) sont incluses dans les attributs renvoyés.

```

const sequelize = require("../database/db");
const { DataTypes } = require('sequelize');
const SaleItem = sequelize.define('saleItem', {
  id: {
    field: 'id',
    type: DataTypes.INTEGER,
    allowNull: false,
    unique: true,
    primaryKey: true,
    autoIncrement: true
  },
  quantity: {
    field: 'quantity',
    type: DataTypes.INTEGER,
    allowNull: false
  },
  price: {
    field: 'price',
    type: DataTypes.FLOAT,
    allowNull: false
  },
  unitprice: {
    type: DataTypes.VIRTUAL,
    get() {
      return this.price/this.quantity;
    },
    set(unitprice){
      throw new Error('Do not try to set the `unitprice` value!');
    }
  }
}, {
  freezeTableName: true,
  timestamps: false
});
module.exports = SaleItem;

```

Après avoir mis à jour le modèle **SaleItem**, accédez à votre fichier **index.js** (où se trouve le serveur Express), et ajoutez la route suivante pour obtenir l'article au prix le plus bas. Testez la route et assurez-vous qu'elle fonctionne.

```

app.get("/items/lowestup", async(req,res)=>{
  try{
    let saleItems = await SaleItem.findAll({
      attributes: ["itemno","price","quantity", "unitprice"]
    });
    // Trier l'objet sequelize renvoyé par prix unitaire croissant.
    saleItems = saleItems.sort(function (a, b) {
      if (a.unitprice < b.unitprice) {
        return -1;
      }
    });
    // Le premier élément est celui qui a le prix unitaire le plus bas,
    // on peut enregistrer la valeur du prix unitaire dans une variable
    let unitprice = saleItems[0].unitprice;
    // en utilisant la méthode findByPk, nous obtenons l'article avec le
    // itemno du premier élément de l'objet renvoyé
    let item = await Item.findByPk(saleItems[0].itemno);
    // nous ajoutons la propriété unitprice à l'instance sequelize avant de
    // l'envoyer à l'utilisateur
    item.setDataValue('unitprice', unitprice);
    return res.status(200).send({status: 1, data: item});
  }catch (e){
    console.error(e);
    return res.status(500).send({status:0, data: "internal error"})
  }
})

```

```

GET localhost:3000/items/lowestup
Params Send Save
Pretty Raw Preview JSON
1 {
2   "status": 1,
3   "data": {
4     "itemno": 3,
5     "itemname": "Compass",
6     "itemtype": "N",
7     "itemcolor": "-",
8     "unitprice": 5
9   }
10 }

```

Exercice 4 : Créez une route qui retourne le client qui a dépensé le plus d'argent. La sortie pourrait ressembler à ceci :

```

GET localhost:3000/clients/spentmost
Params Send Save
Body Cookies Headers (7) Test Results Status: 200 OK Time: 56 ms
Pretty Raw Preview JSON
1 {
2   "status": 1,
3   "data": {
4     "id": 110,
5     "firstname": "Anna",
6     "lastname": "Carolyn",
7     "createdAt": "2022-11-24T11:30:03.712Z",
8     "updatedAt": "2022-11-24T11:30:03.712Z",
9     "totalspendings": 110
10 }

```

```
postgres=# select * from "saleItem";
 id | quantity | price | saleno | itemno
----+-----+-----+-----+-----
  1 |         2 |     10 |       1 |      3
  2 |         3 |     24 |       1 |      1
  3 |         1 |      8 |       2 |      1
  4 |         2 |     10 |       3 |      3
  5 |         3 |     60 |       4 |      5
  6 |         1 |     20 |       4 |      4
  7 |         1 |     10 |       2 |      2
  8 |         2 |     10 |       5 |      3
(8 rows)
```

Total price = 110

le client avec l'identifiant 110 a dépensé le plus d'argent

```
postgres=# select * from sales;
 saleno | saledate | saletext |          createdAt          |          updatedAt          | clientid
-----+-----+-----+-----+-----+-----
  1 | 2020-01-15 | sale 1 | 2022-11-24 12:30:03.74+01 | 2022-11-24 12:30:03.74+01 |    100
  2 | 2020-01-18 | sale 2 | 2022-11-24 12:30:03.74+01 | 2022-11-24 12:30:03.74+01 |    100
  3 | 2020-01-21 | sale 3 | 2022-11-24 12:30:03.74+01 | 2022-11-24 12:30:03.74+01 |    110
  4 | 2020-01-25 | sale 4 | 2022-11-24 12:30:03.74+01 | 2022-11-24 12:30:03.74+01 |    110
  5 | 2020-02-25 | sale 5 | 2022-11-24 12:30:03.74+01 | 2022-11-24 12:30:03.74+01 |    110
(5 rows)
```

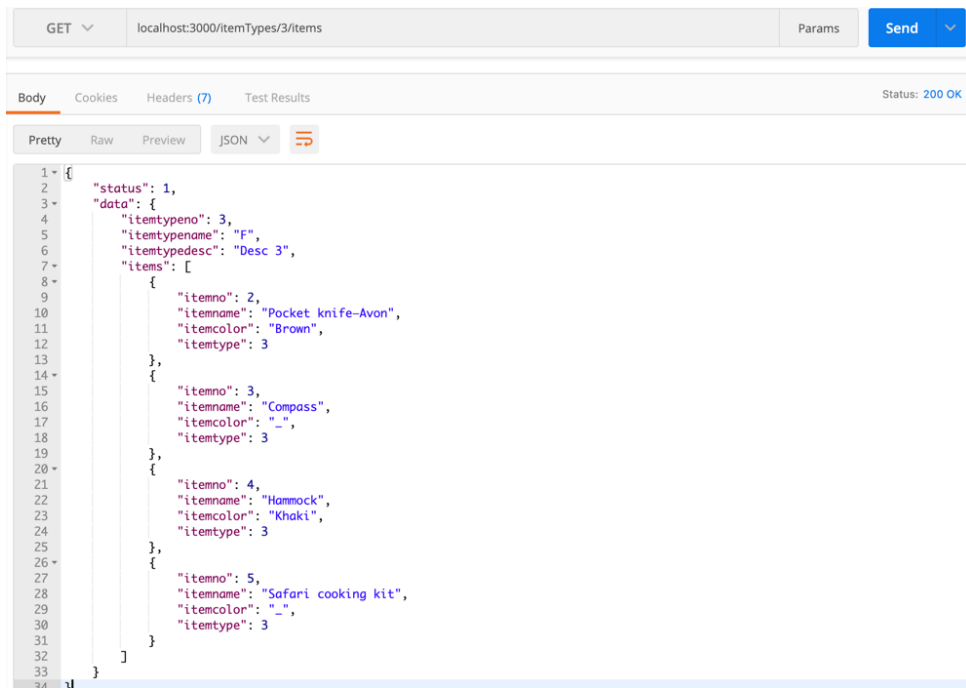
Exercice 5 : Chaque article (**item**) a un type d'article. Lorsque nous avons créé le modèle "**Item**", nous avons défini le type de chaque article (**itemtype**) sous forme de texte codé en dur.

```
postgres=# select * from "item";
 itemno | itemname | itemtype | itemcolor
-----+-----+-----+-----
  1 | Pocket knife-Nile | E | Brown
  2 | Pocket knife-Avon | E | Brown
  3 | Compass | N | -
  4 | Hammock | F | Khaki
  5 | Safari cooking kit | E | -
(5 rows)
```

Une meilleure approche consiste à créer un modèle "**itemType**" séparé et à l'associer au modèle "**item**".

1. Mettez à jour le modèle "**item**" actuel et créez le modèle "**itemType**", puis associez les deux modèles l'un à l'autre.
2. Mettez à jour la fonction `seed_data()`, ajoutez des types d'articles factices et liez-les aux articles.
 - a. Lier un article (**item**) à un type d'article (**itemType**) doit être aléatoire. Vous parcourez les articles et, pour chaque **item**, vous sélectionnez un **itemType** aléatoire dans la base de données (recherchez comment utiliser une fonction random avec Sequelize) et liez le type d'article renvoyé à l'article.
3. Créez la route **GET /itemtypes/:id/items** qui renvoie tous les articles appartenant à un type d'article donné.
4. Créez la route **GET /itemtypes/bestsale** qui renvoie le type d'articles qui a le plus grand nombre d'articles vendus (utilisez le champ de quantité).

Un exemple de sortie de l'exercice 5.3 :



```
1- {
2-   "status": 1,
3-   "data": {
4-     "itemtypeno": 3,
5-     "itemtypename": "F",
6-     "itemtypedesc": "Desc 3",
7-     "items": [
8-       {
9-         "itemno": 2,
10-        "itemname": "Pocket knife-Avon",
11-        "itemcolor": "Brown",
12-        "itemtype": 3
13-      },
14-      {
15-        "itemno": 3,
16-        "itemname": "Compass",
17-        "itemcolor": "-",
18-        "itemtype": 3
19-      },
20-      {
21-        "itemno": 4,
22-        "itemname": "Hammock",
23-        "itemcolor": "Khaki",
24-        "itemtype": 3
25-      },
26-      {
27-        "itemno": 5,
28-        "itemname": "Safari cooking kit",
29-        "itemcolor": "-",
30-        "itemtype": 3
31-      }
32-    ]
33-  }
34- }
```

Un exemple de sortie de l'exercice 5.4 :

GET localhost:3000/itemTypes/bestsale

body

Headers (7)

Test Results

Pretty Raw Preview JSON

```

1 {
2   "status": 1,
3   "data": [
4     {
5       "itemtype": 2,
6       "total_quantity": "1"
7     },
8     {
9       "itemtype": 3,
10      "total_quantity": "5"
11     },
12     {
13       "itemtype": 1,
14       "total_quantity": "9"
15     }
16   ]
17 }

```

(3 rows)

```
postgres=# select * from "itemType";
```

| itemtypeno | itemtypename | itemtypedesc |
|------------|--------------|--------------|
| 1 | E | Desc 1 |
| 2 | N | Desc 2 |
| 3 | F | Desc 3 |

(3 rows)

```
postgres=# select * from "saleItem";
```

| id | quantity | price | saleno | itemno | Type |
|----|----------|-------|--------|--------|----------|
| 1 | 2 | 10 | 1 | 3 | Type = 1 |
| 2 | 3 | 24 | 1 | 1 | Type = 3 |
| 3 | 1 | 8 | 2 | 1 | Type = 3 |
| 4 | 2 | 10 | 3 | 3 | Type = 1 |
| 5 | 3 | 60 | 3 | 5 | Type = 1 |
| 6 | 1 | 20 | 4 | 4 | Type = 2 |
| 7 | 1 | 10 | 4 | 2 | Type = 3 |
| 8 | 2 | 10 | 5 | 3 | Type = 1 |

(8 rows)

Type = 1, SUM = 9
Type = 3, SUM = 5
Type = 2, SUM = 1

```
postgres=# select * from "item";
```

| itemno | itemname | itemcolor | itemtype |
|--------|--------------------|-----------|----------|
| 1 | Pocket knife-Nile | Brown | 3 |
| 2 | Pocket knife-Avon | Brown | 3 |
| 3 | Compass | - | 1 |
| 4 | Hammock | Khaki | 2 |
| 5 | Safari cooking kit | - | 1 |

(5 rows)

GET localhost:3000/itemTypes/bestsale

Authorization Headers (1) Body Pre-request Script Tests

Type No Auth

Body Cookies Headers (7) Test Results

Pretty Raw Preview JSON

```

1 {
2   "status": 1,
3   "data": {
4     "itemtype": 1,
5     "total_quantity": "9"
6   }
7 }

```

Trier et retourner le premier