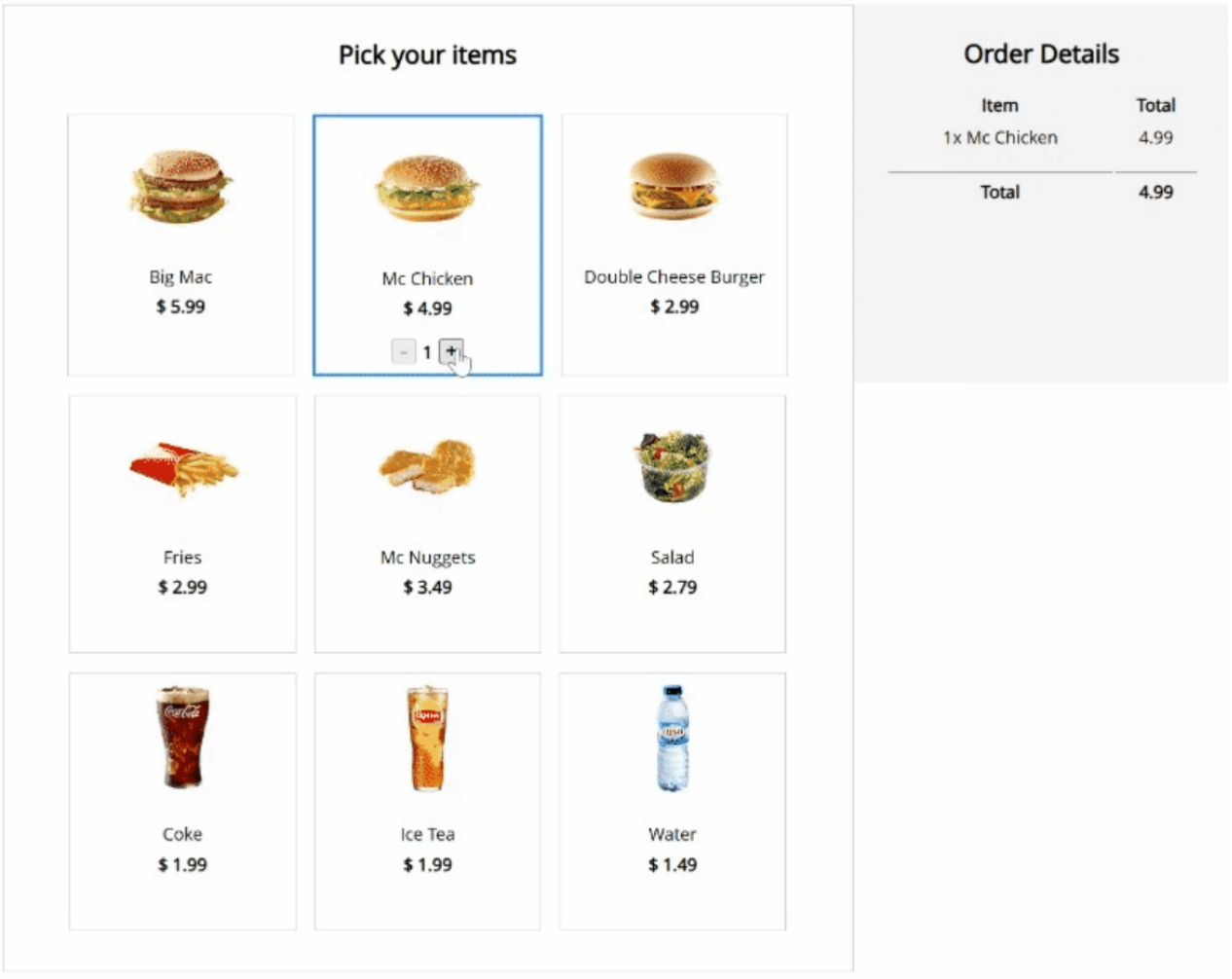
self-service-machine-api - STEP 1

Introduction

Notre but est de réaliser le backend de l'application suivante :



Nous allons utiliser `node.js` et `Express.js`.

Ce backend sera une API REST permettant d'exposer les produits ('Big Mac', 'Mc Chicken', 'Double Cheese Burger', etc) stockés dans une base de données MySQL.

Etape n°1

Dans cette 1ère étape, nous allons :

- configurer le projet
- installer nos premières dépendances
- faire le HelloWorld

Prérequis

Avoir installé la dernière version LTS de node.js

IDE

Nous allons utiliser Visual Studio Code ainsi que l'extension Prettier

Grâce à l'extension prettier, votre code doit être formatté automatiquement lorsque vous faites un `Ctrl+s`.

A vous d'installer et de configurer cette extension !

Initialiser le projet

Pour commencer, nous allons configurer notre projet grâce à la commande suivante :

```
npm init
```

A vous de répondre aux différentes questions.

Ne craignez rien ! Vous pourrez toujours modifier les valeurs par la suite.

Cette commande a créé un fichier `package.json`.

Ce fichier, nous permet de gérer nos dépendances, de définir des commandes, etc

Installer express

Nous allons installer notre 1ère dépendance à savoir `Express.js`.

Il s'agit d'un framework pour node.js qui sera l'outil principal de notre backend.

```
npm install express --save
```

A noter l'option `--save` qui indique que nous aurons besoin de cette dépendance pour faire fonctionner cette application en production.

A chaque fois que nous installons une dépendance, une nouvelle entrée est présente dans le fichier `package.json` et le code de la librairie est placé dans le répertoire `node_modules`.

Installer nodemon

La 2ème dépendance est `nodemon` qui permet de redémarrer "à chaud" notre serveur de développement.

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

A noter l'option `--save-dev` qui indique que nous n'utiliserons cette dépendance que pour le développement (mais pas en production).

Modifier le fichier `package.json` :

Nous allons modifier le fichier `package.json` afin de :

- pouvoir faire un `npm start` pour démarrer l'application (script start)
- pouvoir utiliser la syntaxe des modules ES au lieu de la syntaxe commonJS des modules (l'entrée 'type')

```
{
  "name": "self-service-machine-api",
  "type": "module",
  "version": "1.0.0",
  "description": "API REST to self-service-machine application",
  "main": "src/app.mjs ",
  "scripts": {
    "start": "nodemon src/app.mjs"
  },
  "author": "Charmier Grégory",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2"
  },
  "devDependencies": {
    "nodemon": "^3.0.2"
  }
}
```

Créer un fichier `app.mjs` dans un dossier `src`

Dans ce fichier `app.mjs` nous allons faire le HelloWorld.

Prendre exemple sur https://expressjs.com/fr/starter/hello-world.html pour afficher le HelloWorld.

Attention ! Vous devez modifier les imports en notation CommonJS et les écrire en notation ES.

```
import express from "express";

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

app.get("/", (req, res) => {
  res.send("Hello World!");
});

app.listen(port, () => {
  console.log(`Example app listening on port http://localhost:${port}`);
});
```

Vous pouvez maintenant exécuter l'application : `npm start`

Passons à l'étape n°2

self-service-machine-api - STEP 2

Nous voulons mettre en place notre 1ère route (ou point de terminaison).

Pour l'instant, nous allons simuler la base de données (Mocker) avec un fichier `mock-product.mjs` qui contient un tableau de produits.

Création du fichier `mock-product.mjs` dans le répertoire `db`

```
let products = [
  {
    id: 1,
    name: "Big Mac",
    price: 5.99,
    created: new Date(),
  },
  ...
];

export { products }
```

Création du fichier `products.mjs` dans le répertoire `routes`

Dans ce fichier, on trouvera toutes les routes des produits commençant par '/api/products/'.

La 1ère route est la route permettant de récupérer tous les produits grâce à la requête HTTP :

GET http://localhost:3000/api/products/

```
import express from "express";

import { products } from "../db/mock-product.mjs";

import { success } from "./helper.mjs";

const productsRouter = express();

productsRouter.get("/", (req, res) => {
  const message = "La liste des produits a bien été récupérée.";
  res.json(success(message, products));
});

export { productsRouter };
```

La méthode `success()`

La méthode `success()` a pour but de retourner un objet js qui contient un message et les données au format json.

```
const success = (message, data) => {
  return {
    message: message,
    data: data,
  };
};
```

```
export { success };
```

Retourner du json

Lorsque le consommateur de notre API REST fait un GET http://localhost:3000/api/products/, on doit lui retourner le json des produits.

C'est pour cela que l'on utilise `res.json`

`res` pour response c'est à dire la Réponse HTTP. `json()` la méthode qui retourne l'objet js en json.

Content-Type

Il est important que la réponse HTTP soit bien du json et pas du texte qui ressemble à du json.

Pour cela, il faut s'assurer que la méthode json à bien fait son job.

Un des moyens les plus couramment utilisé est de faire un `curl`.

Lorsque vous pensez que votre code est terminé, faites la commande suivante :

```
curl -i http://localhost:3000/api/products/
```

```
Greg@LAPTOP-5335QTOF MINGW64 ~/OneDrive - Education Vaud/295/self-service-machine-api
$ curl -i http://localhost:3000/api/products
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   765  100   765    0     0  23360      0 --:--:-- --:--:-- --:--:-- 23906HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 765
ETag: W/"2fd-dMhWz5REr8y/c1FeiY7WTEOgsYI"
Date: Wed, 27 Dec 2023 08:56:33 GMT
Connection: keep-alive
Keep-Alive: timeout=5

{"message":"La liste des produits a bien été récupérée.","data":[{"id":1,"name":"Big Mac","price":5.99,"c
reated":"2023-12-27T08:56:17.080Z"},{"id":2,"name":"Mc Chicken","price":4.99,"created":"2023-12-27T08:56:
17.080Z"},{"id":3,"name":"Double Cheese Burger","price":2.99,"created":"2023-12-27T08:56:17.080Z"},{"id":
4,"name":"Fries","price":2.99,"created":"2023-12-27T08:56:17.080Z"},{"id":5,"name":"Mc Nuggets","price":3
.49,"created":"2023-12-27T08:56:17.080Z"},{"id":6,"name":"Salad","price":2.79,"created":"2023-12-27T08:56
:17.080Z"},{"id":7,"name":"Coke","price":1.99,"created":"2023-12-27T08:56:17.080Z"},{"id":8,"name":"Ice T
ea","price":1.99,"created":"2023-12-27T08:56:17.080Z"},{"id":9,"name":"Water","price":1.49,"created":"202
3-12-27T08:56:17.080Z"}]}
```

Comme vous pouvez le voir le `Content-Type` est bien `application/json`

Les points d'entrées / et /api/

Pour terminer, nous avons simplement fait une redirection de '/api/' vers '/'.

Voila le fichier `src/app.mjs` complet :

```
import express from "express";

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

app.get("/", (req, res) => {
  res.send("API REST of self service machine !");
});
```

```
app.get("/api/", (req, res) => {
  res.redirect(`http://localhost:${port}/`);
});

import { productsRouter } from "./routes/products.mjs";
app.use("/api/products", productsRouter);

app.listen(port, () => {
  console.log(`Example app listening on port http://localhost:${port}`);
});
```

Passons à l'étape n°3

self-service-machine-api - STEP 3

Dans cette étape nous allons mettre en place 2 routes :

- la route permettant de récupérer un produit
- la route permettant d'ajouter un nouveau produit

Route GET /api/products/id

Cette route permet de récupérer un produit en particulier.

```
productsRouter.get("/:id", (req, res) => {
  const productId = req.params.id;
  const product = products.find((product) => product.id == productId);
  const message = `Le produit dont l'id vaut ${productId} a bien été récupéré.`;
  res.json(success(message, product));
});
```

Route POST /api/products/

Cette route permet d'ajouter un nouveau produit. C'est probablement la route la plus difficile à mettre en place.

Pour commencer, il faut comprendre que le consommateur de notre API REST va devoir nous envoyer les informations du nouveau produit qu'il souhaite ajouter.

Voila à quoi pourrait ressembler un appel HTTP correspondant en utilisant curl :

```
curl -X POST http://localhost:3000/api/products -H "Content-Type: application/json" -d '{
    "name": "HamburgerVaudois",
    "price": 9.99
}'
```

Le consommateur de notre API REST nous a transmis en json les informations du nouveau produit à ajouter.

A nous de faire le nécessaire pour être capable de prendre en charge une telle requête HTTP.

```
productsRouter.post("/", (req, res) => {
  // Création d'un nouvel id du produit
  // Dans les prochains versions, c'est MySQL qui gérera cela pour nous (identifiant auto_increment)
  const id = getUniqueId(products);

  // Création d'un objet avec les nouvelles informations du produits
  const createdProduct = { ...req.body, ...{ id: id, created: new Date() } };

  // Ajout du nouveau produit dans le tableau
  products.push(createdProduct);

  // Définir un message pour le consommateur de l'API REST
  const message = `Le produit ${createdProduct.name} a bien été créé !`;

  // Retourner la réponse HTTP en json avec le msg et le produit créé
  res.json(success(message, createdProduct));
});
```

Il est important de noter que notre route est `.post()` et plus `.get()` comme précédemment.

Ensuite, nous pouvons commencer à étudier le corps de la méthode.

Nous commençons par générer un nouvel id produit.

```
const id = getUniqueId(products);
```

C'est une fonction créé par nos soins et que je vous conseille de placer dans le fichier `helper.js`.

```
...
const getUniqueId = (products) => {

  // On construit un tableau d'id de produits
  const productsIds = products.map((product) => product.id);

  // La fonction passée à reduce compare deux éléments à la fois (a et b) et
  // retourne le plus grand des deux grâce à Math.max.
  // Au final, reduce parcourt tout le tableau productsIds, compare chaque ID
  // avec l'ID maximal trouvé jusqu'à présent, et retourne l'ID le plus élevé,
  // qui est stocké dans la variable maxId.
  const maxId = productsIds.reduce((a, b) => Math.max(a, b));

  return maxId + 1;
};

export { success, getUniqueId };
```

Ensuite nous créons l'objet produit à utilisant la décomposition d'objet JavaScript.

```
const createdProduct = { ...req.body, ...{ id: id, created: new Date() } };
```

Il nous reste plus qu'à ajouter le nouveau produit à notre liste des produits.

```
products.push(createdProduct);
```

Le reste du code ne devrait pas poser de problème de compréhension.

Nous sommes prêt pour tester notre nouvel route :

```
Greg@LAPTOP-5335QTOF MINGW64 ~/OneDrive - Education Vaud/295/self-se
$ curl -X POST http://localhost:3000/api/products -H "Content-Type:
application/json" -d '{
    "name": "HamburgerVaudois",
    "price": 9.99
}'
  % Total    % Received % Xferd  Average Speed   Time    Time     Ti
me  Current
                                 Dload  Upload   Total   Spent    Le
ft  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:-
100   163  100   110  100    53   5924   2854 --:--:-- --:--:-- --:-
-:--  9588{"message":"Le produit undefined a bien été créé !","data"
:{"id":10,"created":"2023-12-27T10:59:01.758Z"}}
```

Malheureusement, il y a un problème !

Comme on peut le voir le message est : "Le produit undefined a bien été créé !"

Dans notre code, nous utilisons un `req.body` afin de récupérer les données transmises par le consommateur de notre API REST. Malheureusement, `req.body` est `undefined`.

Pourquoi ?

Lorsqu'un consommateur envoie une requête POST avec un corps de requête JSON nous devons convertir le JSON en un objet JavaScript.

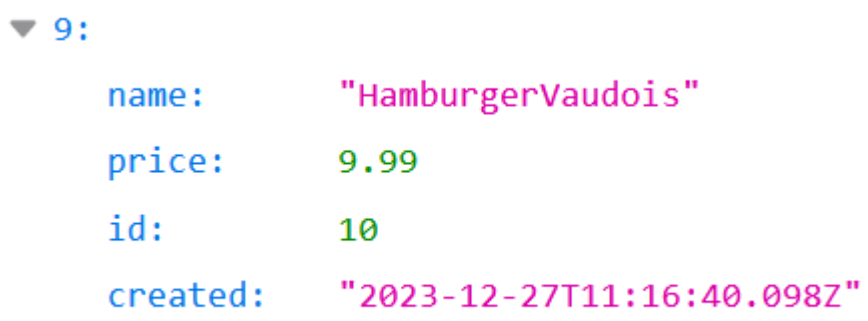
Pour ce faire, nous allons configurer le middleware `express.json()` pour analyser le corps des requêtes JSON.

```
import express from "express";

const app = express();

app.use(express.json());

const port = 3000;

...
```

Si vous testez à nouveau la commande `curl` tout devrait fonctionner maintenant !

D'ailleurs après avoir ajouté votre nouveau produit, vous pouvez refaire un HTTP GET /api/products/ dans votre navigateur.

Un nouveau produit avec l'id 10 devrait être présent :



Passons à l'étape n°4

self-service-machine-api - STEP 4

Dans cette étape nous allons mettre en place 2 routes :

- la route permettant de mettre à jour un produit
- la route permettant de supprimer un produit

Nous allons également apprendre à utiliser un outil qui facilite l'exécution de requêtes HTTP pour tester notre backend.

Route DELETE /api/products/id

```
productsRouter.delete("/:id", (req, res) => {
  const productId = req.params.id;

  let deletedProduct = getProduct(productId);

  removeProduct(productId);

  // Définir un message pour le consommateur de l'API REST
  const message = `Le produit ${deletedProduct.name} a bien été supprimé !`;

  // Retourner la réponse HTTP en json avec le msg et le produit créé
  res.json(success(message, deletedProduct));
});
```

Pour tester le code `curl -X DELETE http://localhost:3000/api/products/1`

```
Greg@LAPTOP-5335QTOF MINGW64 ~/OneDrive - Education Vaud/295/self-service-machine-api (main)
$ curl -X DELETE http://localhost:3000/api/products/1
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   140  100   140    0     0   8561      0 --:--:-- --:--:-- --:--:--  9333{"message":"Le produit Big
Mac a bien été supprimé !","data":{"id":1,"name":"Big Mac","price":5.99,"created":"2023-12-27T11:59:01.23
3Z"}}
```

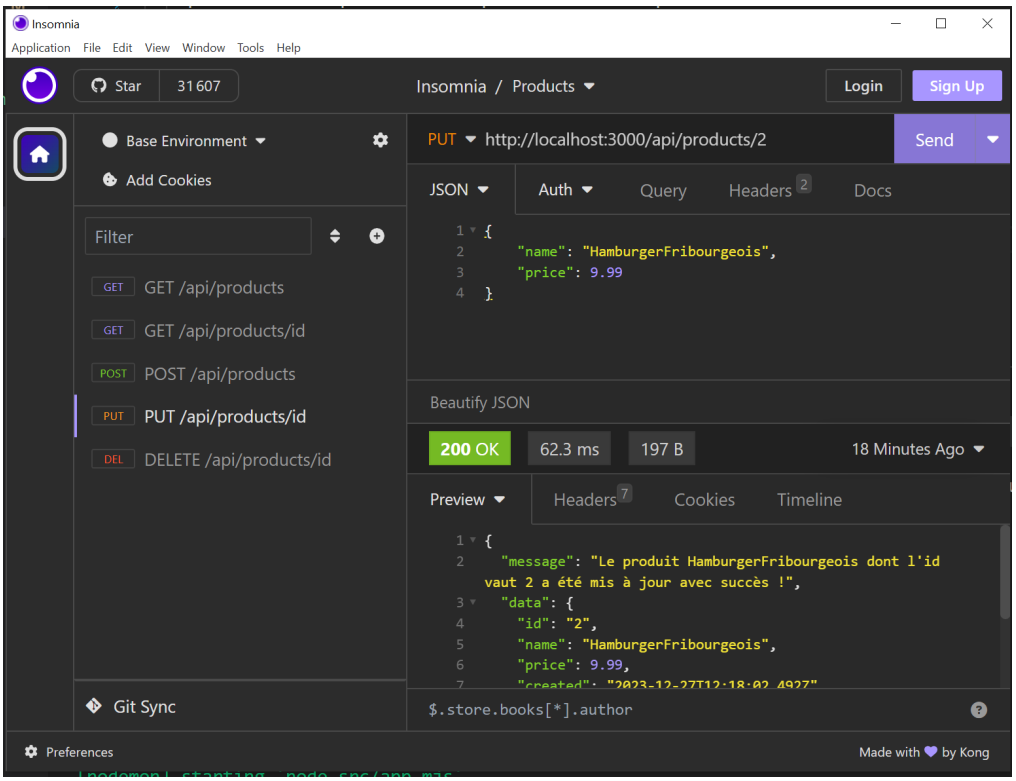
Route PUT /api/products/id

```
productsRouter.put("/:id", (req, res) => {
  const productId = req.params.id;

  const product = getProduct(productId);

  // Mise à jour du produit
  // A noter que la propriété 'created' n'étant pas modifiée, sera conservée telle quelle.
  const updatedProduct = {
    id: productId,
    ...req.body,
    created: product.created,
  };
  updateProduct(productId, updatedProduct);

  // Définir un message pour l'utilisateur de l'API REST
  const message = `Le produit ${updatedProduct.name} dont l'id vaut ${productId} a été mis à jour avec succès !`;

  // Retourner la réponse HTTP en json avec le msg et le produit créé
  res.json(success(message, updatedProduct));
});
```

Nous avons placé certaines fonctions liées aux produits dans le fichier `mock-product.mjs`

```
...

/**
 * Récupère le produit dont l'id vaut `productId`
 * @param {*} productId
 */
const getProduct = (productId) => {
  return products.find((product) => product.id == productId);
};

/**
 * Supprime le produit dont l'id vaut `productId`
 * @param {*} productId
 */
const removeProduct = (productId) => {
  products = products.filter((product) => product.id != productId);
};

/**
 * Met à jour le produit dont l'id vaut `productId`
 * @param {*} productId
 * @param {*} updatedProduct
 */
const updateProduct = (productId, updatedProduct) => {
  products = products.map((product) =>
    product.id == productId ? updatedProduct : product
  );
};

/**
 * Génère et retourne le prochain id des produits
 * @param {*} products
 */
const getUniqueId = () => {
  const productsIds = products.map((product) => product.id);
  const maxId = productsIds.reduce((a, b) => Math.max(a, b));
  const uniqueId = maxId + 1;

  return uniqueId;
};

export { products, getProduct, removeProduct, updateProduct, getUniqueId };
```

Tester notre API REST avec insomnia



Vous trouverez dans le répertoire `/test/insomnia` un export json permettant de tester les différentes routes.

Passons à l'étape n°5

self-service-machine-api - STEP 5

Dans cette étape nous allons mettre en place la base de données !

MySQL

Nous allons utiliser une base de données MySQL.

Comme vous en avez déjà l'habitude nous allons utiliser les 2 conteneurs docker.

Installer l'ORM Sequelize

Nous allons installer l'ORM Sequelize :

`npm install sequelize --save`

Nous allons également installer un driver MySQL :

`npm install mysql2 --save`

Connection à la base de données

Dans un 1er temps, nous devons créé une base de données dans MySQL.

Par exemple, créons la db `db_products`.

Ensuite on créé un fichier `sequelize.mjs` dans le répertoire `db`.

```
import { Sequelize } from "sequelize";

const sequelize = new Sequelize(
  "db_products", // Nom de la DB qui doit exister
  "root", // Nom de l'utilisateur
  "root", // Mot de passe de l'utilisateur
  {
    host: "localhost",
    //port: "6033", pour les conteneurs docker MySQL
    dialect: "mysql",
    logging: false,
  }
);

export { sequelize };
```

Maintenant dans le fichier `src/app.js` :

```
...
const port = 3000;

import { sequelize } from "./db/sequelize.mjs";

sequelize
  .authenticate()
  .then((_) =>
    console.log("La connexion à la base de données a bien été établie")
```

```
  )
  .catch((error) => console.error("Impossible de se connecter à la DB"));

...
```

Maintenant si vous lancez l'application `npm start` vous devriez voir dans la console 'La connexion à la base de données a bien été établie'

Passons à l'étape n°6

self-service-machine-api - STEP 6

Maintenant que nous avons pu nous connecter à notre base de données via l'ORM, nous allons pouvoir commencer à utiliser l'ORM sequelize.

Créer le model Product

Nous allons créer un fichier `products.mjs` dans le dossier `/src/models`

```
// https://sequelize.org/docs/v7/models/data-types/

const ProductModel = (sequelize, DataTypes) => {
  return sequelize.define(
    "Product",
    {
      id: {
        type: DataTypes.INTEGER,
        primaryKey: true,
        autoIncrement: true,
      },
      name: {
        type: DataTypes.STRING,
        allowNull: false,
      },
      price: {
        type: DataTypes.FLOAT,
        allowNull: false,
      },
    },
    {
      timestamps: true,
      createdAt: "created",
      updatedAt: false,
    }
  );
};

export { ProductModel };
```

Synchroniser les données de MOCK avec la DB

Nous pouvons maintenant utiliser le modèle `ProductModel` pour créer une relation 1 <--> 1 avec la table `products` de la base de données.

De plus, pour avoir quelques produits dans cette table, nous allons utiliser les produits présents dans le fichier `mock-product.mjs` et les importer dans la DB.

Le code ci-dessous doit être présent dans le fichier `src/db/sequelize.mjs`.

```
import { Sequelize, DataTypes } from "sequelize";
import { ProductModel } from "../models/products.mjs";

const sequelize = new Sequelize(
  "db_products", // Nom de la DB qui doit exister
  "root", // Nom de l'utilisateur
  "root", // Mot de passe de l'utilisateur
  {
    host: "localhost",
    dialect: "mysql",
```

```
    logging: false,
  }
);

import { products } from "./mock-product.mjs";

// Le modèle product
const Product = ProductModel(sequelize, DataTypes);

let initDb = () => {
  return sequelize
    .sync({ force: true }) // Force la synchro => donc supprime les données également
    .then((_) => {
      importProducts();
      console.log("La base de données db_products a bien été synchronisée");
    });
};

const importProducts = () => {
  // import tous les produits présents dans le fichier db/mock-product
  products.map((product) => {
    Product.create({
      name: product.name,
      price: product.price,
    }).then((product) => console.log(product.toJSON()));
  });
};

export { sequelize, initDb, Product };
```

Il nous reste plus qu'à appeler la méthode `initDb()` dans le fichier `src/app.mjs`

```
...
sequelize
  .authenticate()
  .then((_) =>
    console.log("La connexion à la base de données a bien été établie")
  )
  .catch((error) => console.error("Impossible de se connecter à la DB"));

initDb();
...
```

Voila ! Les produits ont tous été importés dans la table `products` de la DB.

D'ailleurs vous devriez les voir dans la console du serveur.

Passons à l'étape n°7

self-service-machine-api - STEP 7

Maintenant nous allons pouvoir modifier nos routes afin que ces dernières utilisent l'ORM sequelize plutôt qu'une liste statique de produits.

Sequelize a quelques méthodes de base que nous allons utiliser :

- Model.create()
- Model.findAll()
- Model.findByPk()
- Model.update()
- Model.destroy()

Comme toujours, la documentation officielle doit être consultée :

https://sequelize.org/docs/v6/core-concepts/model-querying-basics/

Modification des routes

Route GET /api/products/

Pour récupérer tous les produits présents en base de données, nous allons utiliser la méthode `findAll()` de l'ORM `sequelize`.

Pour ce faire, nous allons appeler cette méthode `findAll()` sur le modèle `Product` créé dans l'étape précédente.

Il est important de comprendre que chaque méthode de l'ORM Sequelize est une promesse javascript.

Définition d'une promesse :

Une promesse est un objet (Promise) qui représente la complétion (la réussite) ou l'échec d'une opération asynchrone.

Vous devez donc bien connaître cette notion js pour comprendre le code que nous allons écrire maintenant :

Théorie sur les promesses en JS

Voilà ce que cela donne dans le code du fichier `routes/products.mjs` :

```
productsRouter.get("/", (req, res) => {
  Product.findAll().then((products) => {
    const message = "La liste des produits a bien été récupérée.";
    res.json(success(message, products));
  });
});
```

Route GET /api/products/:id

Pour récupérer un produit dont l'id est passé en paramètre, nous allons cette fois utiliser la méthode `findByPk()` sur le modèle `Product`. Cette méthode prend en paramètre la clé primaire (Primary Key (Pk) en anglais) du produit que l'on souhaite récupérer.

Voilà ce que cela donne dans le code du fichier `routes/products.mjs` :

```
productsRouter.get("/:id", (req, res) => {
  Product.findByPk(req.params.id).then((product) => {
```

```
    const message = `Le produit dont l'id vaut ${product.id} a bien été récupéré.`;
    res.json(success(message, product));
  });
});
```

Route POST /api/products/

Pour l'ajout d'un nouveau produit, nous allons utiliser la méthode `create()` en passant en paramètre les données fournies par l'utilisateur. Comme d'habitude, on récupère les données dans le corps de la requête HTTP grâce à `req.body`.

Vous vous souvenez que grâce au middleware `express.json()` le json fourni a été transformé en javascript !

Attention ! Ce n'est pas une bonne pratique de ne pas vérifier ou valider les données envoyées par l'utilisateur. Mais pour l'instant on commence par faire fonctionner notre API REST avec l'ORM `Sequilize` et la DB `MySQL`. Dans une prochaine étape, nous allons ajouter de la validation sur les données du client.

Voilà ce que cela donne dans le code du fichier `routes/products.mjs` :

```
productsRouter.post("/", (req, res) => {
  Product.create(req.body).then((createdProduct) => {
    // Définir un message pour le consommateur de l'API REST
    const message = `Le produit ${createdProduct.name} a bien été créé !`;

    // Retourner la réponse HTTP en json avec le msg et le produit créé
    res.json(success(message, createdProduct));
  });
});
```

Route DELETE /api/products/:id

Pour la suppression d'un produit, nous allons utiliser 2 méthodes de l'ORM :

La 1ère est la méthode `findByPk()` que nous allons déjà utiliser. Nous allons commencer par récupérer le produit que nous souhaitons supprimer. Ainsi nous pourrons afficher des informations sur ce produit supprimé à notre consommateur (notre client).

La 2ème méthode est la méthode `destroy()` qui permet de supprimer le produit en question. La syntaxe est un peu différente des méthodes utilisées jusqu'à présent. En effet, nous devons fournir l'id du produit à supprimer mais à une propriété `where`.

Un dernier point concernant l'enchaînement des promesses. En effet, nous avons une 1ère promesse pour la récupération du produit à supprimer. Ensuite une 2ème promesse pour supprimer le produit.

Nous avons donc 2 `then` que s'enchaînent.

Le principe du chaînage des promesses est une notion importante en JS que vous devez maîtriser.

```
productsRouter.delete("/:id", (req, res) => {
  Product.findByPk(req.params.id).then((deletedProduct) => {
    Product.destroy({
      where: { id: deletedProduct.id },
    }).then((_) => {
      // Définir un message pour le consommateur de l'API REST
      const message = `Le produit ${deletedProduct.name} a bien été supprimé !`;

      // Retourner la réponse HTTP en json avec le msg et le produit créé
      res.json(success(message, deletedProduct));
    });
```

```
  });
});
```

Route PUT /api/products/:id

Pour la mise à jour d'un produit nous allons commencer par utiliser la méthode `update()` en lui passant :

- le `req.body`
- l'id du produit à mettre à jour

Comme pour la suppression, il y a un chaînage des promesses.

```
productsRouter.put("/:id", (req, res) => {
  const productId = req.params.id;
  Product.update(req.body, { where: { id: productId } }).then((_) => {
    Product.findByPk(productId).then((updatedProduct) => {
      // Définir un message pour l'utilisateur de l'API REST
      const message = `Le produit ${updatedProduct.name} dont l'id vaut ${updatedProduct.id} a été mis à jour avec succè

      // Retourner la réponse HTTP en json avec le msg et le produit créé
      res.json(success(message, updatedProduct));
    });
  });
});
```

Ci-dessous, vous trouverez le code complet pour le fichier `routes/products.mjs`

```
import express from "express";

import { Product } from "../db/sequelize.mjs";

import { success } from "./helper.mjs";

const productsRouter = express();

productsRouter.get("/", (req, res) => {
  Product.findAll().then((products) => {
    const message = "La liste des produits a bien été récupérée.";
    res.json(success(message, products));
  });
});

productsRouter.get("/:id", (req, res) => {
  Product.findByPk(req.params.id).then((product) => {
    const message = `Le produit dont l'id vaut ${product.id} a bien été récupéré.`;
    res.json(success(message, product));
  });
});

productsRouter.post("/", (req, res) => {
  Product.create(req.body).then((createdProduct) => {
    // Définir un message pour le consommateur de l'API REST
    const message = `Le produit ${createdProduct.name} a bien été créé !`;

    // Retourner la réponse HTTP en json avec le msg et le produit créé
    res.json(success(message, createdProduct));
  });
});

productsRouter.delete("/:id", (req, res) => {
  Product.findByPk(req.params.id).then((deletedProduct) => {
    Product.destroy({
      where: { id: deletedProduct.id },
```

```
    }).then((_) => {
      // Définir un message pour le consommateur de l'API REST
      const message = `Le produit ${deletedProduct.name} a bien été supprimé !`;

      // Retourner la réponse HTTP en json avec le msg et le produit créé
      res.json(success(message, deletedProduct));
    });
  });
});

productsRouter.put("/:id", (req, res) => {
  const productId = req.params.id;
  Product.update(req.body, { where: { id: productId } }).then((_) => {
    Product.findByPk(productId).then((updatedProduct) => {
      // Définir un message pour l'utilisateur de l'API REST
      const message = `Le produit ${updatedProduct.name} dont l'id vaut ${updatedProduct.id} a été mis à jour avec succè

      // Retourner la réponse HTTP en json avec le msg et le produit créé
      res.json(success(message, updatedProduct));
    });
  });
});

export { productsRouter };
```

Suppression du code mort

Après avoir vérifier que votre API fonctionne à nouveau en testant vos routes avec Insomnia, il nous reste à supprimer le code mort.

Dans le fichier `mock-product.mjs` on peut supprimer toutes les fonctions :

```
...

/**
 * Récupère le produit dont l'id vaut `productId`
 * @param {*} productId
 */
const getProduct = (productId) => {
  return products.find((product) => product.id == productId);
};

/**
 * Supprime le produit dont l'id vaut `productId`
 * @param {*} productId
 */
const removeProduct = (productId) => {
  products = products.filter((product) => product.id != productId);
};

/**
 * Met à jour le produit dont l'id vaut `productId`
 * @param {*} productId
 * @param {*} updatedProduct
 */
const updateProduct = (productId, updatedProduct) => {
  products = products.map((product) =>
    product.id == productId ? updatedProduct : product
  );
};

/**
 * Génère et retourne le prochain id des produits
 * @param {*} products
 */
const getUniqueId = () => {
```

```
  const productsIds = products.map((product) => product.id);
  const maxId = productsIds.reduce((a, b) => Math.max(a, b));
  const uniqueId = maxId + 1;

  return uniqueId;
};

...
```

C'était une étape importante ! Notre API REST fonctionne maintenant avec une base de données !
Passons à l'étape n°8

self-service-machine-api - STEP 8

Maintenant nous allons nous intéresser à la gestion des erreurs.

Selon l'erreur que le consommateur va rencontrer, on veut que le statut HTTP retourné correspond à l'erreur en question.

Les statuts HTTP

Voici la liste complète des status HTTP : https://developer.mozilla.org/fr/docs/Web/HTTP/Status

A noter le statut 418, mon préféré : `I'm a teapot` (je suis une théière) 😉

Gestion du statut HTTP 404 - Création de notre 1er middleware

Si un consommateur de notre API REST tente d'utiliser une URL non définie, nous devons nous assurer que :

- le statut d'erreur est bien `404`
- un message d'erreur lui indique clairement que cette URL ne correspond à aucune ressource pour notre API REST.

Pour ce faire, nous allons créer notre 1er middleware !

En effet, nous avons déjà utiliser des middlewares mais nous n'en avons pas encore créé un.

Allons-y !

Dans notre fichier `app.mjs` après la définition de nos routes, vous pouvez ajouter le code suivant :

```
...
// Si aucune route ne correspondant à l'URL demandée par le consommateur
app.use(({ res }) => {
  const message =
    "Impossible de trouver la ressource demandée ! Vous pouvez essayer une autre URL.";
  res.status(404).json(message);
});
...
```

`app.use` est utilisée pour ajouter des middlewares à notre application Express.

Les middlewares sont des fonctions qui ont accès aux objets de requête (`req`), de réponse (`res`). Le `app.use` est utilisé ici sans chemin spécifique, donc ce middleware sera exécuté pour toutes les requêtes.

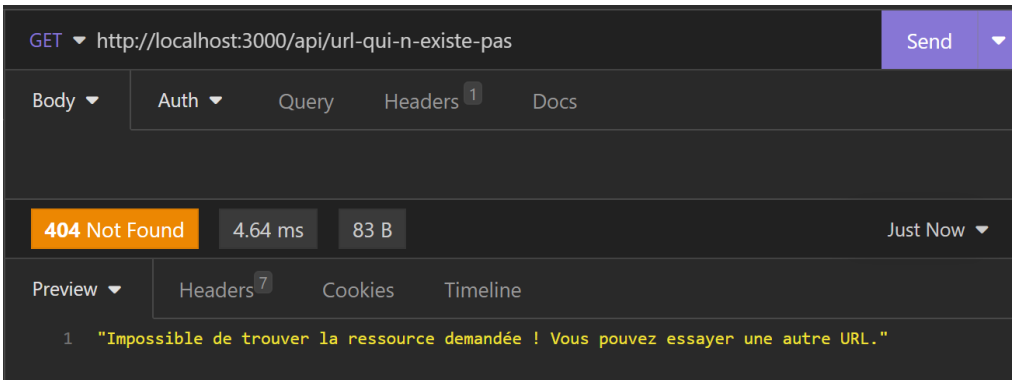
`({ res }) => { ... }` est une fonction fléchée (Arrow function) en JavaScript, et elle utilise la déstructuration pour extraire `res` de l'objet argument. Cela signifie que cette fonction prend un objet en entrée qui a au moins une propriété `res` , qui est l'objet de réponse Express

`res.status(404)` définit le code d'état de la réponse HTTP à 404. Le code 404 est standard pour "Not Found" (Non trouvé), ce qui signifie que le serveur n'a pas trouvé ce qui était demandé.

Ce middleware est exécuté lorsqu'aucune autre route n'a été trouvée correspondant à la requête entrante. C'est pourquoi ce code placé à la fin de toutes les autres définitions de route. Il envoie une réponse avec le statut 404 et un message JSON indiquant à l'utilisateur que la ressource demandée n'a pas été trouvée.

Test du middleware pour la gestion du statut HTTP 404 avec insomnia

Maintenant si avec insomnia nous tentons accéder à une URL qui ne correspond à rien pour notre API REST, nous pouvons voir que le statut et le message sont corrects.



HTTP 500 pour la route GET /api/products

Pour la route GET /api/products, nous devons gérer le cas où un problème surviendrait lors de la promesse. Dans ce cas, on doit retourner un statut HTTP 500 et indiqué un message d'erreur adéquat.

```
...
productsRouter.get("/", (req, res) => {
  Product.findAll()
    .then((products) => {
      const message = "La liste des produits a bien été récupérée.";
      res.json(success(message, products));
    })
    .catch((error) => {
      const message =
        "La liste des produits n'a pas pu être récupérée. Merci de réessayer dans quelques instants.";
      res.status(500).json({ message, data: error });
    });
});
...
```

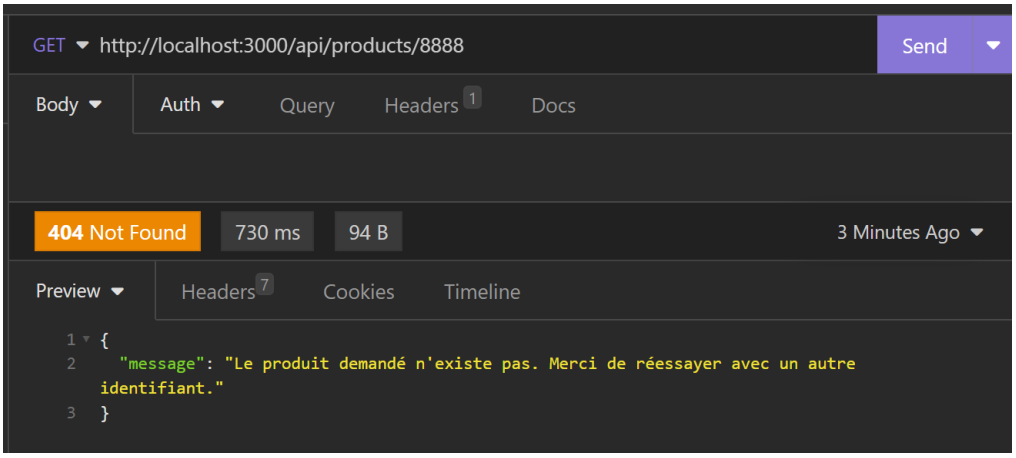
Maintenant si nous faisons volontairement une erreur (par exemple ajoutons un s à products => productss) dans le code puis avec insomnia nous tentons accéder à GET /api/products/ nous pouvons voir que le statut 500 et le message d'erreur.

HTTP 500 et HTTP 404 pour la route GET /api/products/:id

Pour la route GET /api/products/:id, nous devons gérer les cas :

- où un problème surviendrait lors de la promesse comme précédemment.
- mais également le cas où le consommateur a demandé un produit avec un id qui n'existe pas.

```
...

productsRouter.get("/:id", (req, res) => {
  Product.findByPk(req.params.id)
    .then((product) => {
      if (product === null) {
        const message =
          "Le produit demandé n'existe pas. Merci de réessayer avec un autre identifiant.";
        // A noter ici le return pour interrompre l'exécution du code
        return res.status(404).json({ message });
      }
      const message = `Le produit dont l'id vaut ${product.id} a bien été récupéré.`;
      res.json(success(message, product));
    })
    .catch((error) => {
      const message =
        "Le produit n'a pas pu être récupéré. Merci de réessayer dans quelques instants.";
      res.status(500).json({ message, data: error });
    });
});
...
```



HTTP 500 pour la route POST /api/products/

Pour la route POST /api/products/, nous devons gérer le cas où un problème surviendrait lors de la promesse comme précédemment.

```
...
productsRouter.post("/", (req, res) => {
  Product.create(req.body)
    .then((createdProduct) => {
      // Définir un message pour le consommateur de l'API REST
      const message = `Le produit ${createdProduct.name} a bien été créé !`;

      // Retourner la réponse HTTP en json avec le msg et le produit créé
      res.json(success(message, createdProduct));
    })
```

```
    .catch((error) => {
      const message =
        "Le produit n'a pas pu être ajouté. Merci de réessayer dans quelques instants.";
      res.status(500).json({ message, data: error });
    });
});
...
```

HTTP 500 et HTTP 404 pour la route PUT /api/products/:id

Comme dans le cas de l'update nous utilisons :

- `Product.update()`
- `Product.findByPk()`

nous avons 2 cas de HTTP 500 à gérer.

De même, comme nous utilisons `findByPk()` le consommateur a pu renseigné un id qui n'existe pas. Nous avons donc un HTTP 404 à gérer.

```
...
productsRouter.put("/:id", (req, res) => {
  const productId = req.params.id;
  Product.update(req.body, { where: { id: productId } })
    .then((_) => {
      Product.findByPk(productId)
        .then((updatedProduct) => {
          if (updatedProduct === null) {
            const message =
              "Le produit demandé n'existe pas. Merci de réessayer avec un autre identifiant.";
            // A noter ici le return pour interrompre l'exécution du code
            return res.status(404).json({ message });
          }
          // Définir un message pour l'utilisateur de l'API REST
          const message = `Le produit ${updatedProduct.name} dont l'id vaut ${updatedProduct.id} a été mis à jour avec s

          // Retourner la réponse HTTP en json avec le msg et le produit créé
          res.json(success(message, updatedProduct));
        })
        .catch((error) => {
          const message =
            "Le produit n'a pas pu être mis à jour. Merci de réessayer dans quelques instants.";
          res.status(500).json({ message, data: error });
        });
    })
    .catch((error) => {
      const message =
        "Le produit n'a pas pu être mis à jour. Merci de réessayer dans quelques instants.";
      res.status(500).json({ message, data: error });
    });
});
...
```

HTTP 500 pour la route DELETE /api/products/:id

Nous laissons volontairement de côté cette route pour l'instant.

Car nous n'en avons pas encore terminé avec la gestion des statuts HTTP.

Nous allons continuer et améliorer cela dans l'étape 9 (step9).

Passons à l'étape n°9

self-service-machine-api - STEP 9

Dans cette étape nous allons terminer la gestion des statuts HTTP.

Amélioration du code pour la route PUT /api/products/:id

Dans notre code, nous avons du code dupliqué.

En effet, nous avons 2 fois le code :

```
...
.catch((error) => {
      const message =
        "Le produit n'a pas pu être mis à jour. Merci de réessayer dans quelques instants.";
      res.status(500).json({ message, data: error });
    });
...
```

Nous allons corriger cela en ne "catchant" une éventuelle erreur que dans le `catch` le plus englobant.

Pour faire cela, il faut bien comprendre que nous devons ajouter un `return` devant Product.findByPk(). Ainsi en cas d'erreur dans le corps de la méthode Product.findByPk() une erreur sera levée et comme `Product.findByPk()` retourne une promesse, elle sera transmise au bloc parent.

Voici le code corrigé :

```
...
productsRouter.put("/:id", (req, res) => {
  const productId = req.params.id;
  Product.update(req.body, { where: { id: productId } })
    .then((_) => {
      return Product.findByPk(productId).then((updatedProduct) => {
        if (updatedProduct === null) {
          const message =
            "Le produit demandé n'existe pas. Merci de réessayer avec un autre identifiant.";
          // A noter ici le return pour interrompre l'exécution du code
          return res.status(404).json({ message });
        }
        // Définir un message pour l'utilisateur de l'API REST
        const message = `Le produit ${updatedProduct.name} dont l'id vaut ${updatedProduct.id} a été mis à jour avec su

        // Retourner la réponse HTTP en json avec le msg et le produit créé
        res.json(success(message, updatedProduct));
      });
    })
    .catch((error) => {
      const message =
        "Le produit n'a pas pu être mis à jour. Merci de réessayer dans quelques instants.";
      res.status(500).json({ message, data: error });
    });
});
...
```

La route DELETE /api/products/:id

Rien de nouveau dans cette section.

Ce sera simplement um bon exercice pour voir si vous avez compris ce que nous avons fait précédemment.

```
...
productsRouter.delete("/:id", (req, res) => {
  Product.findByPk(req.params.id)
    .then((deletedProduct) => {
      if (deletedProduct === null) {
        const message =
          "Le produit demandé n'existe pas. Merci de réessayer avec un autre identifiant.";
        // A noter ici le return pour interrompre l'exécution du code
        return res.status(404).json({ message });
      }
      return Product.destroy({
        where: { id: deletedProduct.id },
      }).then((_) => {
        // Définir un message pour le consommateur de l'API REST
        const message = `Le produit ${deletedProduct.name} a bien été supprimé !`;

        // Retourner la réponse HTTP en json avec le msg et le produit créé
        res.json(success(message, deletedProduct));
      });
    })
    .catch((error) => {
      const message =
        "Le produit n'a pas pu être supprimé. Merci de réessayer dans quelques instants.";
      res.status(500).json({ message, data: error });
    });
});
...
```

Passons à l'étape n°10

self-service-machine-api - STEP 10

Dans cette étape, nous allons mettre en place la validation des valeurs envoyées par le consommateur de notre API REST.

Pour cela, nous allons maintenant modifier notre modèle `ProductModel` .

Les 1ères règles de validation

Nous voulons nous assurer que :

- le nom ne soit composé que de lettres et du caractère espace.
- le prix soit un float.
- le nom et le prix soient renseignés (non null et non vide)

Voila les changements dans `models/products.mjs`

```
// https://sequelize.org/docs/v7/models/data-types/

const ProductModel = (sequelize, DataTypes) => {
  return sequelize.define(
    "Product",
    {
      id: {
        type: DataTypes.INTEGER,
        primaryKey: true,
        autoIncrement: true,
      },
      name: {
        type: DataTypes.STRING,
        allowNull: false,
        validate: {
          is: {
            args: /^[A-Za-z\s]*$/,
            msg: "Seules les lettres et les espaces sont autorisées.",
          },
          notEmpty: {
            msg: "Le nom ne peut pas être vide.",
          },
          notNull: {
            msg: "Le nom est une propriété obligatoire.",
          },
        },
      },
      price: {
        type: DataTypes.FLOAT,
        allowNull: false,
        validate: {
          isFloat: {
            msg: "Utilisez uniquement des nombres pour le prix.",
          },
          notEmpty: {
            msg: "Le prix ne peut pas être vide.",
          },
          notNull: {
            msg: "Le prix est une propriété obligatoire.",
          },
        },
      },
    },
    {
      timestamps: true,
      createdAt: "created",
      updatedAt: false,
```

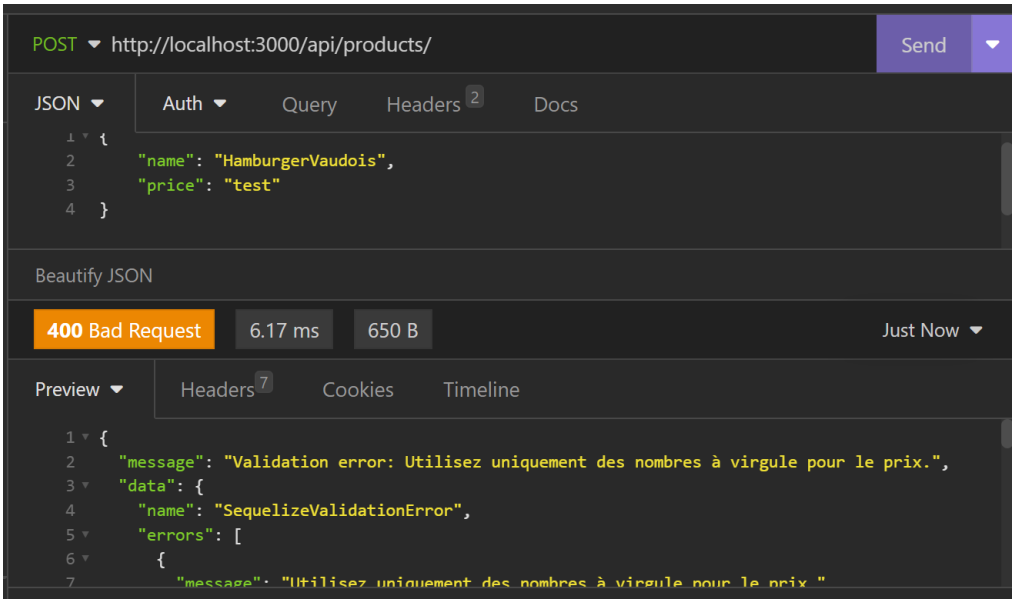
```
    }
  );
};

export { ProductModel };
```

Pour la route POST /api/products/ on a regarde si l'erreur est une instance de `ValidationError`. Si c'est le cas, on récupère et on affiche l'erreur personnalisée.

```
productsRouter.post("/", (req, res) => {
  Product.create(req.body)
    .then((createdProduct) => {
      // Définir un message pour le consommateur de l'API REST
      const message = `Le produit ${createdProduct.name} a bien été créé !`;

      // Retourner la réponse HTTP en json avec le msg et le produit créé
      res.json(success(message, createdProduct));
    })
    .catch((error) => {
      if (error instanceof ValidationError) {
        return res.status(400).json({ message: error.message, data: error });
      }
      const message =
        "Le produit n'a pas pu être ajouté. Merci de réessayer dans quelques instants.";
      res.status(500).json({ message, data: error });
    });
});
```

Maintenant si on fait une requête HTTP POST /api/products avec le json suivant

```
{
  "name": "HamburgerVaudois",
  "price": "test"
}
```

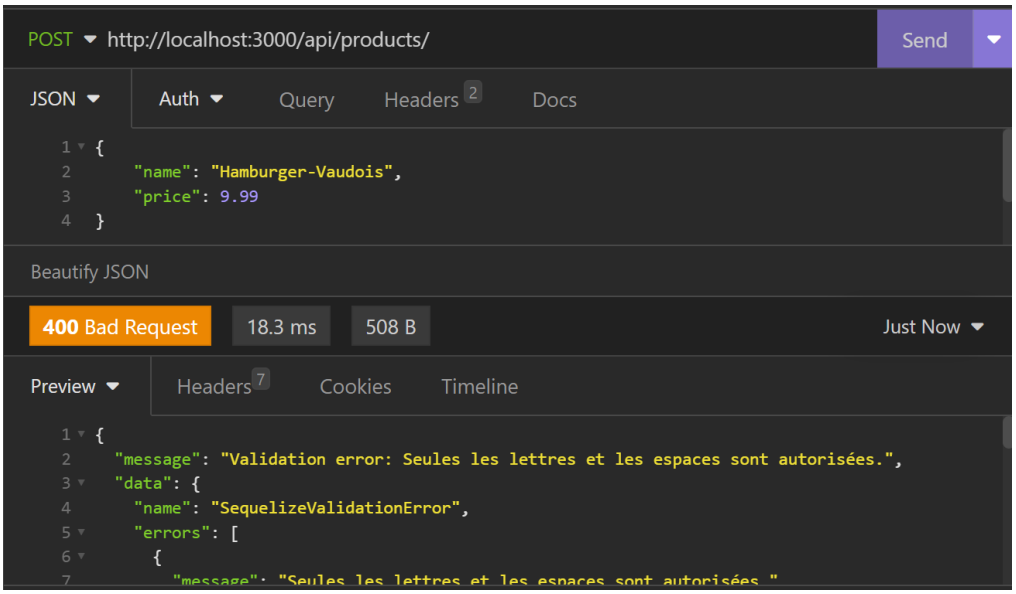


De même, si on fait une requête HTTP POST /api/products avec le json suivant :

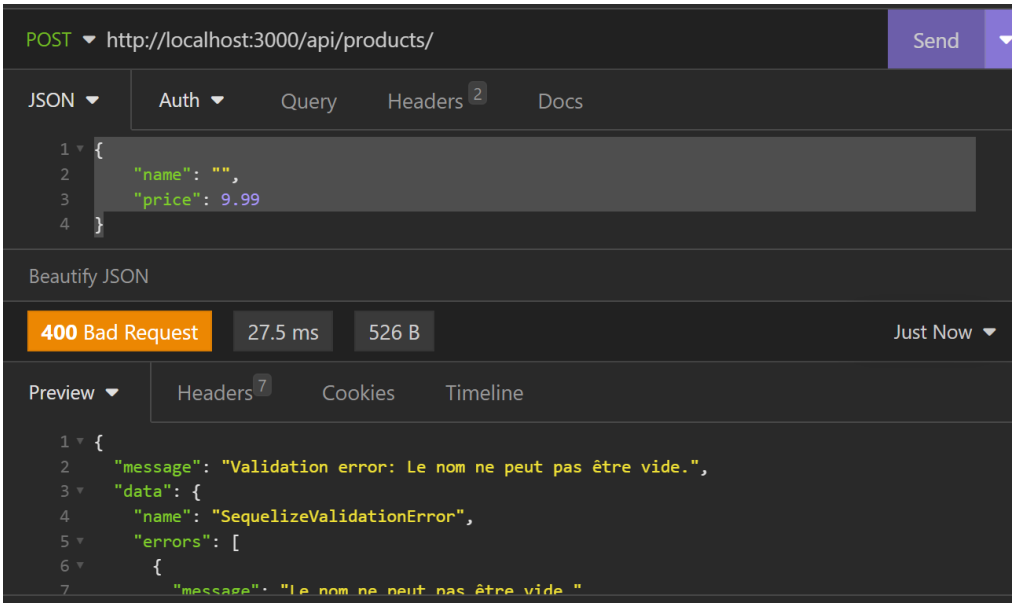
```
{
  "name": "Hamburger-Vaudois",
```

```
  "price": 9.99
}
```

ou encore avec un nom vide :




```
{
  "name": "",
  "price": 9.99
}
```



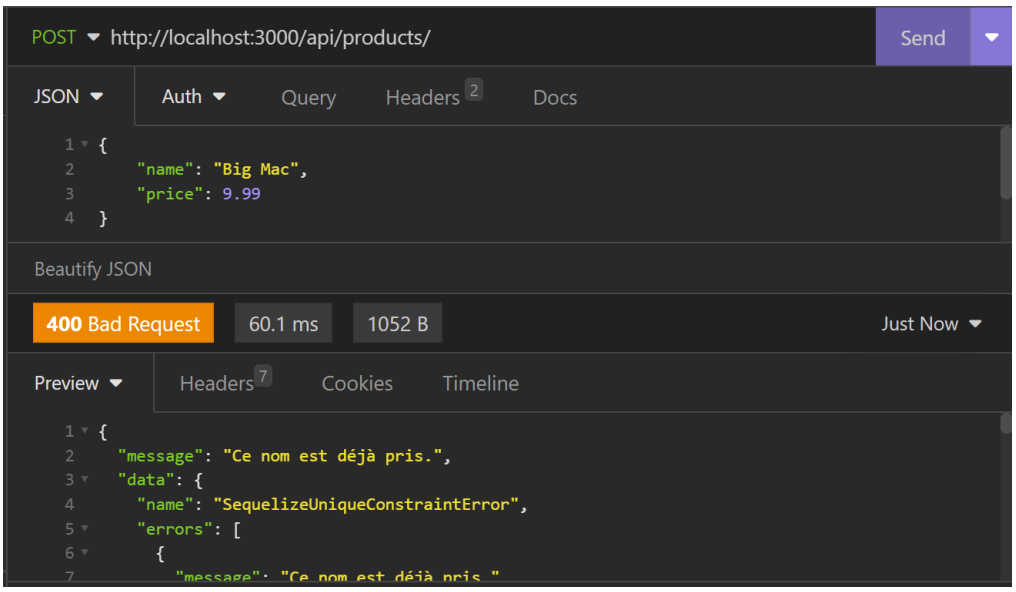
Les autres règles de validation

Nous voulons que :

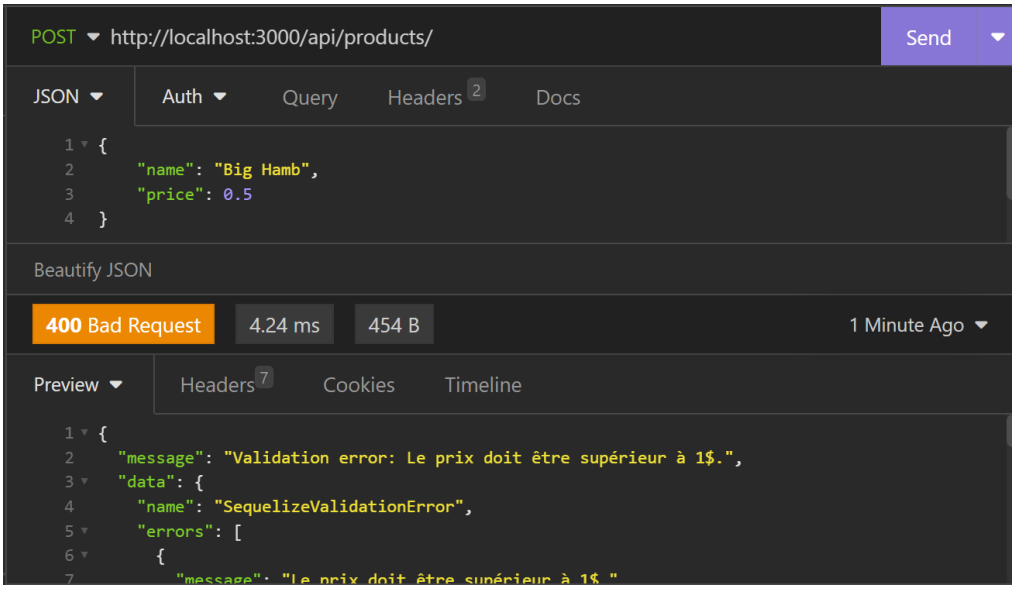
- le nom soit unique
- le prix soit supérieur à 1$ et inférieur à 1000$

```
// https://sequelize.org/docs/v7/models/data-types/

const ProductModel = (sequelize, DataTypes) => {
  return sequelize.define(
    "Product",
    {
      id: {
        type: DataTypes.INTEGER,
        primaryKey: true,
        autoIncrement: true,
      },
      name: {
        type: DataTypes.STRING,
        allowNull: false,
        unique: {
          msg: "Ce nom est déjà pris.",
        },
        validate: {
          is: {
            args: /^[A-Za-z\s]*$/,
            msg: "Seules les lettres et les espaces sont autorisées.",
          },
          notEmpty: {
            msg: "Le nom ne peut pas être vide.",
          },
          notNull: {
            msg: "Le nom est une propriété obligatoire.",
          },
        },
      },
      price: {
        type: DataTypes.FLOAT,
        allowNull: false,
        validate: {
          isFloat: {
            msg: "Utilisez uniquement des nombres pour le prix.",
          },
          notEmpty: {
            msg: "Le prix ne peut pas être vide.",
          },
          notNull: {
            msg: "Le prix est une propriété obligatoire.",
          },
          min: {
            args: [1.0],
            msg: "Le prix doit être supérieur à 1$.",
          },
          max: {
            args: [1000.0],
            msg: "Le prix doit être inférieur à 1000$.",
          },
        },
      },
    },
    {
      timestamps: true,
      createdAt: "created",
      updatedAt: false,
    }
  );
};

export { ProductModel };
```

Il nous reste à tester que la validation pour le nom unique fonctionne :



et que la validation pour (par exemple) le prix inférieur à 1$ fonctionne également :



Passons à l'étape n°11

self-service-machine-api - STEP 11

Mise en place de la recherche

Nous allons maintenant améliorer notre API REST en ajouter une recherche à travers nos produits.

Pour cela nous allons modifier le code de la route GET /api/products.

En effet, nous allons apprendre à utiliser les paramètres de requêtes GET.

On commence par ajouter dans le fichier `mock-product.mjs` le produit suivant :

```
...

{
  id: 10,
  name: "Big Tasty",
  price: 5.99,
  created: new Date(),
},

...
```

Maintenant nous disposons de 2 produits avec le mot 'big'.

Nous aimerions faire une recherche à travers pour les produits en basant cette recherche sur le nom du produit.

GET /api/products?name=big

Nous aimerions que cela retourne tous les produits qui contiennent le mot 'big'.

Voila le code qui permet de faire cela :

```
...
import { ValidationError, Op } from "sequelize";
...

productsRouter.get("/", (req, res) => {
  if (req.query.name) {
    return Product.findAll({
      where: { name: { [Op.like]: `%${req.query.name}%` } },
    }).then((products) => {
      const message = `Il y a ${products.length} produits qui correspondent au terme de la recherche`;
      res.json(success(message, products));
    });
  }
  Product.findAll()
    .then((products) => {
      const message = "La liste des produits a bien été récupérée.";
      res.json(success(message, products));
    })
    .catch((error) => {
      const message =
        "La liste des produits n'a pas pu être récupérée. Merci de réessayer dans quelques instants.";
      res.status(500).json({ message, data: error });
    });
});
```

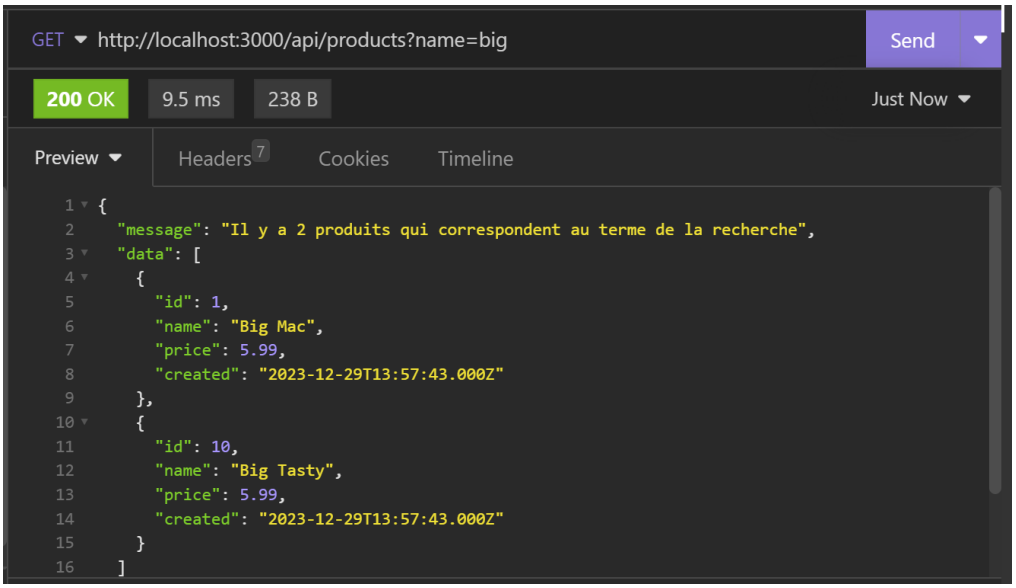
```
...
```

Vous connaissez déjà l'opérateur LIKE utilisé dans le code ci-dessus. En effet si nous devions faire cette requête en SQL, nous ferions :

```
SELECT *
FROM products
WHERE name LIKE '%big%';
```

C'est exactement ce que nous faisons mais en utilisant l'ORM sequelize.

Nous pouvons vérifier que ce code fonctionne bien :



Mise en place d'une limitation pour la recherche

Avec une grande quantité de produits, chaque recherche via l'appel HTTP GET /api/products?name=… qui retournent un certain nombre de produits pourrait mettre un temps conséquent à s'exécuter.

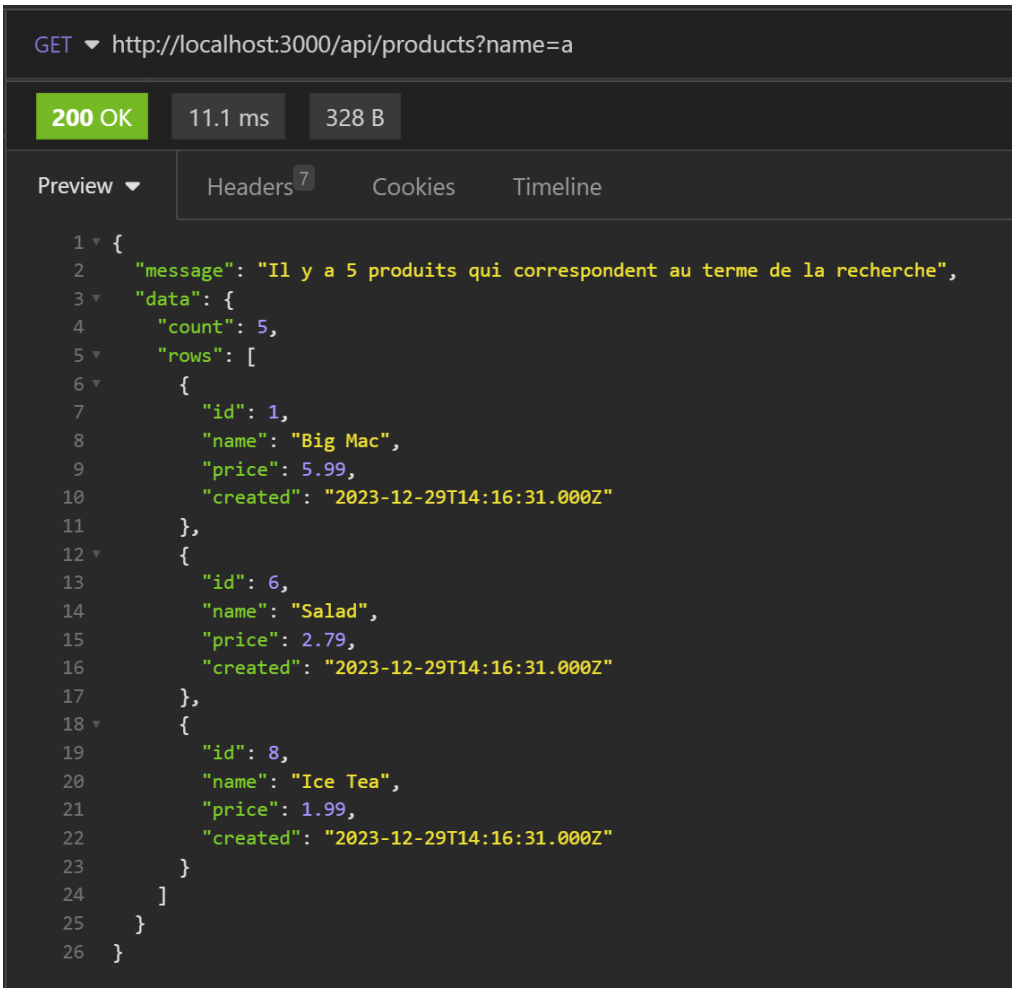
Nous allons donc limiter le nombre de résultats de manière arbitraire à 3 pour le moment. Nous rendrons cela dynamique par la suite.

Nous allons également vouloir afficher le nombre total de produits qui correspondent à la recherche même si à cause de la limitation, le consommateur ne verra que les 3 premiers pour l'instant.

```
...
productsRouter.get("/", (req, res) => {
  if (req.query.name) {
    return Product.findAndCountAll({
      where: { name: { [Op.like]: `%${req.query.name}%` } },
      limit: 3,
    }).then((products) => {
      const message = `Il y a ${products.count} produits qui correspondent au terme de la recherche`;
      res.json(success(message, products));
    });
  }
  Product.findAll()
    .then((products) => {
```

```
      const message = "La liste des produits a bien été récupérée.";
      res.json(success(message, products));
    })
    .catch((error) => {
      const message =
        "La liste des produits n'a pas pu être récupérée. Merci de réessayer dans quelques instants.";
      res.status(500).json({ message, data: error });
    });
});
...
```

Nous pouvons vérifier que ce code fonctionne bien :



Nous indiquons bien que 5 produits correspondent à la recherche même si nous limiter à 3 le résultat de la recherche.

Exercice :

Le consommateur de notre API REST a besoin de choisir le nombre de produits qu'il souhaite. Par exemple dans le cas d'une pagination.

A vous de rendre cette limitation dynamique c'est à dire que pour le cas de la recherche, le consommateur peut saisir l'URL suivante : GET /api/products?name=a&limit=1

A vous de mettre en place le code permettant de prendre en charge ce nouveau paramètre GET limit.

Solution de l'exercice :

```
productsRouter.get("/", (req, res) => {
  if (req.query.name) {
    let limit = 3;
    if (req.query.limit) {
      limit = parseInt(req.query.limit);
    }
    return Product.findAndCountAll({
      where: { name: { [Op.like]: `%${req.query.name}%` } },
      limit: limit,
    }).then((products) => {
      const message = `Il y a ${products.count} produits qui correspondent au terme de la recherche`;
      res.json(success(message, products));
    });
  }
  Product.findAll()
    .then((products) => {
      const message = "La liste des produits a bien été récupérée.";
      res.json(success(message, products));
    })
    .catch((error) => {
      const message =
        "La liste des produits n'a pas pu être récupérée. Merci de réessayer dans quelques instants.";
      res.status(500).json({ message, data: error });
    });
});
```

Passons à l'étape n°12

self-service-machine-api - STEP 12

Dans cette étape nous allons :

- ordonner les résultats par odre alphabétique
- améliorer la recherche

Ordonner le résultat par ordre alphabétique

Commençons par ordonner le résultat de la route GET /api/products

```
...
Product.findAll({ order: ["name"] })
...
```

En ajoutant simplement `{ order: ["name"] }` nous faisons l'équivalent SQL de `ORDER BY name`.

Voila le code complet :

```
...
productsRouter.get("/", (req, res) => {
  if (req.query.name) {
    let limit = 3;
    if (req.query.limit) {
      limit = parseInt(req.query.limit);
    }
    return Product.findAndCountAll({
      where: { name: { [Op.like]: `%${req.query.name}%` } },
      order: ["name"],
      limit: limit,
    }).then((products) => {
      const message = `Il y a ${products.count} produits qui correspondent au terme de la recherche`;
      res.json(success(message, products));
    });
  }
  Product.findAll({ order: ["name"] })
    .then((products) => {
      const message = "La liste des produits a bien été récupérée.";
      res.json(success(message, products));
    })
    .catch((error) => {
      const message =
        "La liste des produits n'a pas pu être récupérée. Merci de réessayer dans quelques instants.";
      res.status(500).json({ message, data: error });
    });
});
...
```

Vous pouvez vérifier que le résultat est bien ordonné sur le nom du produit à la fois pour le résultat de la recherche mais également pour le résultat de tous les produits.

Améliorer la recherche

Si le critère de la recherche ne contient que 1 seul caractère, nous n'avons pas envie d'exécuter une recherche qui risque :

- d'être "gourmande" au niveau des performances

- de ne pas être bien utile pour notre consommateur

```
...
productsRouter.get("/", (req, res) => {
  if (req.query.name) {
    if (req.query.name.length < 2) {
      const message = `Le terme de la recherche doit contenir au moins 2 caractères`;
      return res.status(400).json({ message });
    }
    let limit = 3;
    if (req.query.limit) {
      limit = parseInt(req.query.limit);
    }
    return Product.findAndCountAll({
      where: { name: { [Op.like]: `%${req.query.name}%` } },
      order: ["name"],
      limit: limit,
    }).then((products) => {
      const message = `Il y a ${products.count} produits qui correspondent au terme de la recherche`;
      res.json(success(message, products));
    });
  }
  Product.findAll({ order: ["name"] })
    .then((products) => {
      const message = "La liste des produits a bien été récupérée.";
      res.json(success(message, products));
    })
    .catch((error) => {
      const message =
        "La liste des produits n'a pas pu être récupérée. Merci de réessayer dans quelques instants.";
      res.status(500).json({ message, data: error });
    });
});
...
```

Voila ce que cela donne :

Passons à l'étape n°13

self-service-machine-api - STEP 13

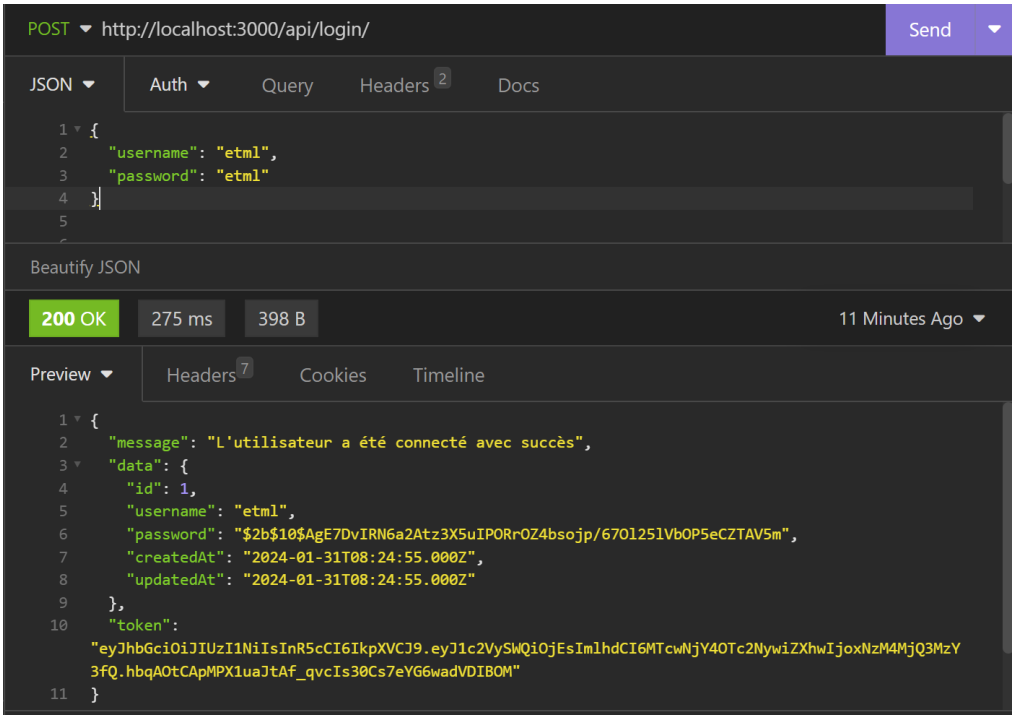
Nous allons mettre en place un système d'authentification.

L'authentification est un processus qui permet de restreindre l'accès aux points de terminaison (aux routes) de notre API REST.

Cette authentification comprend 2 étapes :

- Lors de la **1ère étape**, un utilisateur se connecte à l'aide son nom d'utilisateur et son mot de passe à l'API REST.

Dans notre cas, l'utilisateur utilise la route `POST /api/login/` en fournissant un json comprenant son `username` et son `password`.



A la suite de cette authentification, l'utilisateur obtient un jeton JWT.

En effet, l'authentification entre une application web et une API REST repose sur un standard reconnu, qui est l'authentification avec les « JSON Web Token ».

Chaque client doit obtenir un identifiant unique appelé un jeton JWT.

Un jeton JWT a une durée de validité limitée dans le temps, et se présente sous la forme d'une chaîne de caractères.

Le jeton JWT transite dans l'en-tête HTTP authorization, avec pour valeur "Bearer ".

- La **2ème étape** est l'utilisation de ce jeton JWT :

Une fois que l'utilisateur a obtenu le jeton JWT, il devra l'utiliser à chaque requête vers l'API REST. Ainsi les échanges entre le consommateur et l'API REST seront sécurisés.

2 exigences au niveau de la sécurité

La mise en place d'une authentification côté API REST nécessite de respecter deux exigences principales au niveau de la sécurité :

- chiffrer le mot de passe de l'utilisateur
- sécuriser l'échange des données entre le consommateur et l'API REST par l'utilisation de jeton JWT

Création d'un modèle UserModel

Pour commencer, nous avons besoin de stocker des utilisateurs dans la base de données. Nous allons donc créer un modèle `UserModel`.

```
const UserModel = (sequelize, DataTypes) => {
  return sequelize.define("User", {
    id: {
      type: DataTypes.INTEGER,
      primaryKey: true,
      autoIncrement: true,
    },
    username: {
      type: DataTypes.STRING,
      allowNull: false,
      unique: { msg: "Ce username est déjà pris." },
    },
    password: {
      type: DataTypes.STRING,
      allowNull: false,
    },
  });
};

export { UserModel };
```

Rien de nouveau pour ce modèle.

La table MySQL coorespondante à ce modèle est la table `User`.

A noter que l'on s'assure que le `username` soit unique.

Maintenant que la table `User` existe, nous devons importer des utilisateurs.

Importer un utilisateur en base de données

Le mot de passe de chaque utilisateur ne doit pas être stocké en clair pour des questions de sécurité.

Nous aurons besoin de la dépendance `bcrypt` pour hasher le mot de passe de l'utilisateur.

Comme nous en avons maintenant l'habitude, pour installer la dépendance, nous pouvons utiliser la commande suivante :

`npm install bcrypt --save`

Voila le code permettant d'ajouter un utilisateur `etml` avec un mot de passe hashé.

Le mot de passe en clair est `etml`.

```
const importUsers = () => {
  bcrypt
    .hash("etml", 10) // temps pour hasher = du sel
```

```
    .then((hash) =>
      User.create({
        username: "etml",
        password: hash,
      })
    )
    .then((user) => console.log(user.toJSON()));
};
```

Dans ce code, le seul élément nouveau est l'utilisation de la fonction `hash` permettant de hasher le mot de passe `etml`.

Nous pouvons appeler cette méthode `importUsers` juste après l'importation des produits.

```
let initDb = () => {
  return sequelize
    .sync({ force: true }) // Force la synchro => donc supprime les données également
    .then((_) => {
      importProducts();
      importUsers();
      console.log("La base de données db_products a bien été synchronisée");
    });
};
```

On peut vérifier en base de données que l'utilisateur est bien créé :

```
1 • SELECT * FROM users;
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

id	username	password	createdAt	updatedAt
1	etml	$2b$10$14AxURQ6JwW...	2023-12-30 09:57:47	2023-12-30 09:57:47
NULL	NULL	NULL	NULL	NULL

Authentification de l'utilisateur

Nous devons mettre en place la nouvelle route permettant à un utilisateur de se connecter.

Nous allons isoler cette route dans un fichier js `src/routes/login.mjs`.

Et dans le fichier `src/app.mjs` nous pouvons importer le router du login.

```
...
import { productsRouter } from "./routes/products.mjs";
app.use("/api/products", productsRouter);

import { loginRouter } from "./routes/login.mjs";
app.use("/api/login", loginRouter);
...
```

Nous devons maintenant construire la route permettant à un utilisateur de s'authentifier à l'aide de son nom d'utilisateur et de son mot de passe et ainsi pouvoir obtenir le jeton JWT.

Pour cela nous avons besoin de la librairie `jsonwebtoken` pour générer un jeton JWT.

```
npm install jsonwebtoken --save
```

Voila le code complet de la nouvelle route `POST /api/login/`.

```
import express from "express";
import bcrypt from "bcrypt";
import jwt from "jsonwebtoken";

import { User } from "../db/sequelize.mjs";
import { privateKey } from "../auth/private_key.mjs";

const loginRouter = express();

loginRouter.post("/", (req, res) => {
  User.findOne({ where: { username: req.body.username } })
    .then((user) => {
      if (!user) {
        const message = `L'utilisateur demandé n'existe pas`;
        return res.status(404).json({ message });
      }
      bcrypt
        .compare(req.body.password, user.password)
        .then((isPasswordValid) => {
          if (!isPasswordValid) {
            const message = `Le mot de passe est incorrecte.`;
            return res.status(401).json({ message });
          } else {
            // JWT
            const token = jwt.sign({ userId: user.id }, privateKey, {
              expiresIn: "1y",
            });
            const message = `L'utilisateur a été connecté avec succès`;
            return res.json({ message, data: user, token });
          }
        });
    })
    .catch((error) => {
      const message = `L'utilisateur n'a pas pu être connecté. Réessayez dans quelques instants`;
      return res.json({ message, data: error });
    });
});

export { loginRouter };
```

Nous allons détailler 2 portions de code contenant des éléments nouveaux.

Comparaison des mots de passe

```
bcrypt.compare(req.body.password, user.password);
```

Ce code permet de comparer le mot de passe hashé de l'utilisateur stocké dans la base de données avec le mot en clair fourni dans le json pour notre consommateur.

On utilise la méthode `compare()` de la librairie `bcrypt`.

Générer un jeton JWT

Pour générer un jeton JWT, on utilise la méthode `sign` de la librairie `jsonwebtoken`.

```
const token = jwt.sign({ userId: user.id }, privateKey, {
  expiresIn: "1y",
```

```
});
```

Cette méthode prend :

- En 1er paramètre le payload. Ici un objet js contenant l'id de l'utilisateur.
- En 2ème paramètre une clé privée. Cette clé privée est simplement ici une chaîne de caractères. Il ne faut jamais la partager. Attention ! Même pas dans github !

```
const privateKey = "CUSTOM_PRIVATE_KEY";

export { privateKey };
```

- En 3ème paramètre un objet js pour définir des options. Dans notre cas, on précise dans combien de temps va expirer ce jeton JWT. "1y" signifie un an (1 year).

Définition du payload :

En développement web, le terme "payload" fait généralement référence aux données transmises entre le serveur web et le navigateur client dans le cadre des requêtes HTTP (Hypertext Transfer Protocol) et des réponses correspondantes. Plus précisément, il s'agit de la partie des données de la réponse HTTP qui contient le contenu réel que le navigateur affiche à l'utilisateur.

Utilisation du jeton JWT

Maintenant que le consommateur de notre API REST a obtenu son jeton JWT, il doit l'utiser pour chaque requête.

Nous devons donc récupérer ce jeton JWT et vérifier sa validité.

Dans chacune des routes des produits, nous allons ajouter un nouveau paramètre en 2ème position.

Dans le fichier `src/routes/products.mjs` :

```
...
import { auth } from "../auth/auth.mjs";

const productsRouter = express();

productsRouter.get("/", auth, (req, res) => {
...
```

TODO : A vous de faire cela pour toutes les routes !

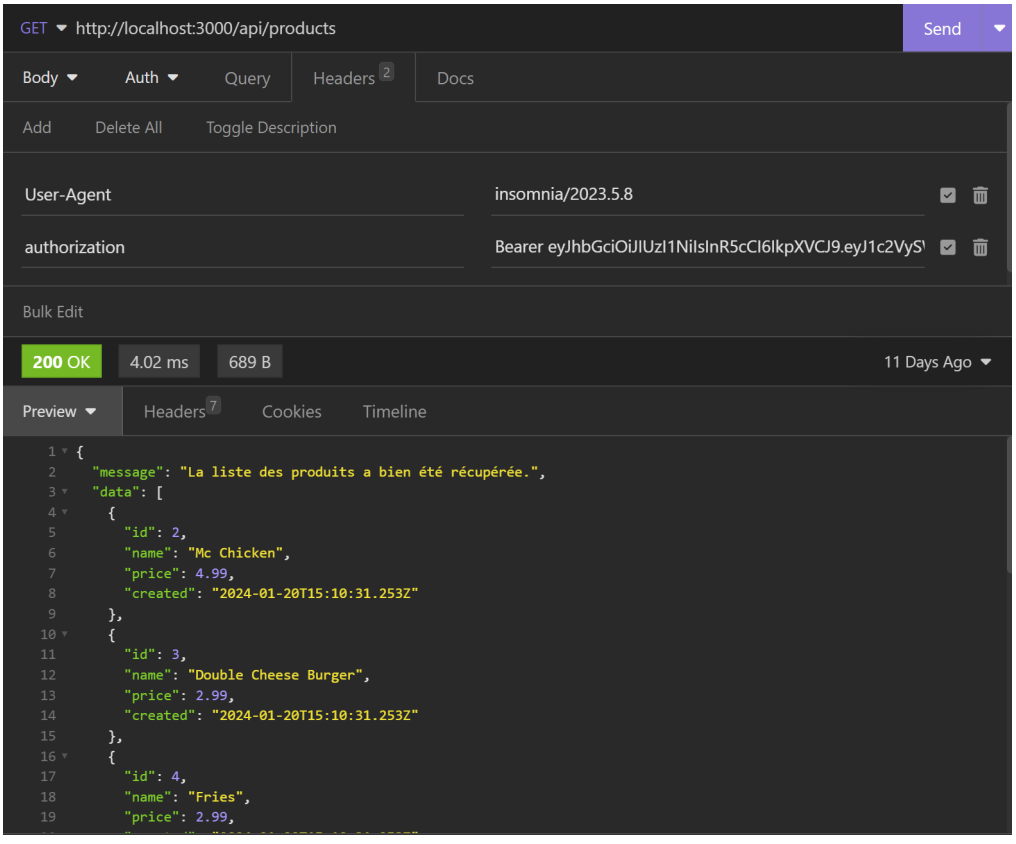
C'est maintenant à nous d'implémenter le fichier `src/auth/auth.mjs` afin de vérifier le jeton JWT fourni par le consommateur.

```
import jwt from "jsonwebtoken";
import { privateKey } from "./private_key.mjs";

const auth = (req, res, next) => {
  const authorizationHeader = req.headers.authorization;

  if (!authorizationHeader) {
    const message = `Vous n'avez pas fourni de jeton d'authentification. Ajoutez-en un dans l'en-tête de la requête.`;
    return res.status(401).json({ message });
  } else {
```

```
    const token = authorizationHeader.split(" ")[1];
    const decodedToken = jwt.verify(
      token,
      privateKey,
      (error, decodedToken) => {
        if (error) {
          const message = `L'utilisateur n'est pas autorisé à accéder à cette ressource.`;
          return res.status(401).json({ message, data: error });
        }
        const userId = decodedToken.userId;
        if (req.body.userId && req.body.userId !== userId) {
          const message = `L'identifiant de l'utisateur est invalide`;
          return res.status(401).json({ message });
        } else {
          next();
        }
      }
    );
  }
};

export { auth };
```

TODO : A vous de commenter en détail le code ci-dessus afin de le comprendre parfaitement

Maintenant que le code est en place, vous devez tester toutes vos routes mais cette fois en utilisant un jeton JWT.



Cette étape était fondamentale afin de sécuriser notre API REST !

Dans la prochaine étape, nous allons nous intéresser à la documentation de notre API REST.

Passons à l'étape n°14

self-service-machine-api - STEP 14

Documentation Swagger

Il est important de documenter notre API REST afin d'aider les consommateurs de notre API REST.

Installation des dépendances

Nous allons commencer par installer 2 dépendances :

```
npm install swagger-jsdoc swagger-ui-express --save
```

Configuration

Ensuite voila le code pour la configuration :

```
// src/swagger.mjs

import swaggerJSDoc from "swagger-jsdoc";

const options = {
  definition: {
    openapi: "3.0.0",
    info: {
      title: "API self-service-machine",
      version: "1.0.0",
      description:
        "API REST permettant de gérer l'application self-service-machine",
    },
    servers: [
      {
        url: "http://localhost:3000",
      },
    ],
    components: {
      securitySchemes: {
        bearerAuth: {
          type: "http",
          scheme: "bearer",
          bearerFormat: "JWT",
        },
      },
      schemas: {
        Teacher: {
          type: "object",
          required: ["name", "price", "created"],
          properties: {
            id: {
              type: "integer",
              description: "L'identifiant unique du produit.",
            },
            name: {
              type: "string",
              description: "Le nom du produit.",
            },
            price: {
              type: "float",
              description: "Le prix du produit.",
            },
```

```
            created: {
              type: "string",
              format: "date-time",
              description: "La date et l'heure de l'ajout d'un produit.",
            },
          },
        },
        // Ajoutez d'autres schémas ici si nécessaire
      },
    },
    security: [
      {
        bearerAuth: [],
      },
    ],
  },
  apis: ["./src/routes/*.mjs"], // Chemins vers vos fichiers de route
};

const swaggerSpec = swaggerJSDoc(options);

export { swaggerSpec };
```

Utilisation de la configuration

Maintenant, il faut utiliser cette configuration dans `src/app.mjs`

```
...
import swaggerUi from "swagger-ui-express";
...

const port = 3000;

import { swaggerSpec } from "./swagger.mjs";

// Route pour accéder à la documentation Swagger
//const specs = swaggerJsdoc(options);
app.use(
  "/api-docs",
  swaggerUi.serve,
  swaggerUi.setup(swaggerSpec, { explorer: true })
);
```

Documentation de notre première route

Ensuite, nous pouvons commencer à définir la documentation d'une route, par exemple, la route GET /api/products/

```
...

/**
 * @swagger
 * /api/products/:
 *   get:
 *     tags: [Products]
 *     security:
 *       - bearerAuth: []
 *     summary: Retrieve all products.
 *     description: Retrieve all products. Can be used to populate a select HTML tag.
 *     responses:
 *       200:
 *         description: All products.
 *         content:
```

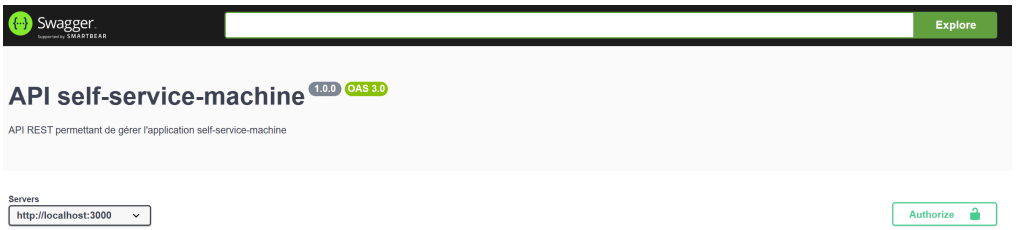
```
 *           application/json:
 *             schema:
 *               type: object
 *               properties:
 *                 data:
 *                   type: object
 *                   properties:
 *                     id:
 *                       type: integer
 *                       description: The product ID.
 *                       example: 1
 *                     name:
 *                       type: string
 *                       description: The product's name.
 *                       example: Big Mac
 *                     price:
 *                       type: number
 *                       description: The product's price.
 *                       example: 5.99
 *
 */
productsRouter.get("/", auth, (req, res) => {
...
```

Tester l'api en fournissant un token JWT

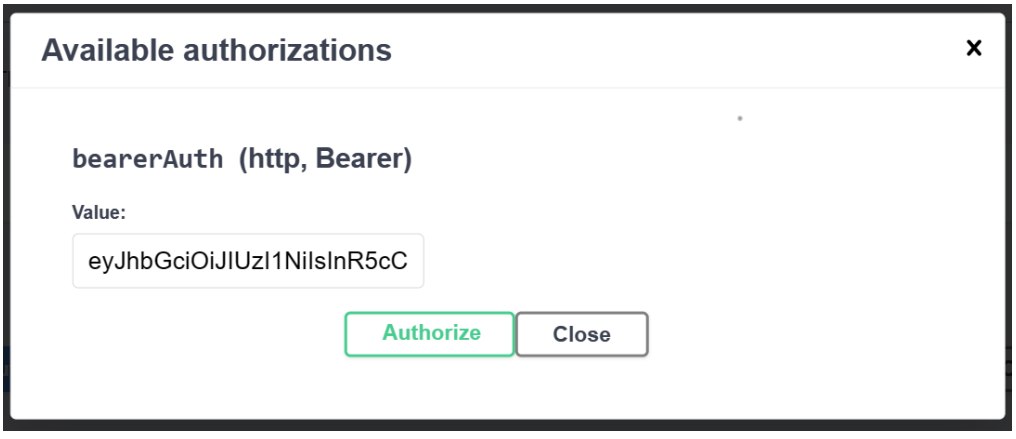
Notre API REST est protégé.

Donc seuls les consommateurs possédant un jeton JWT peuvent effectuer un appel HTTP via la documentation.

Pour cela, un consommateur doit commencer par cliquer sur le bouton `Authorize` (voir ci-dessous) :



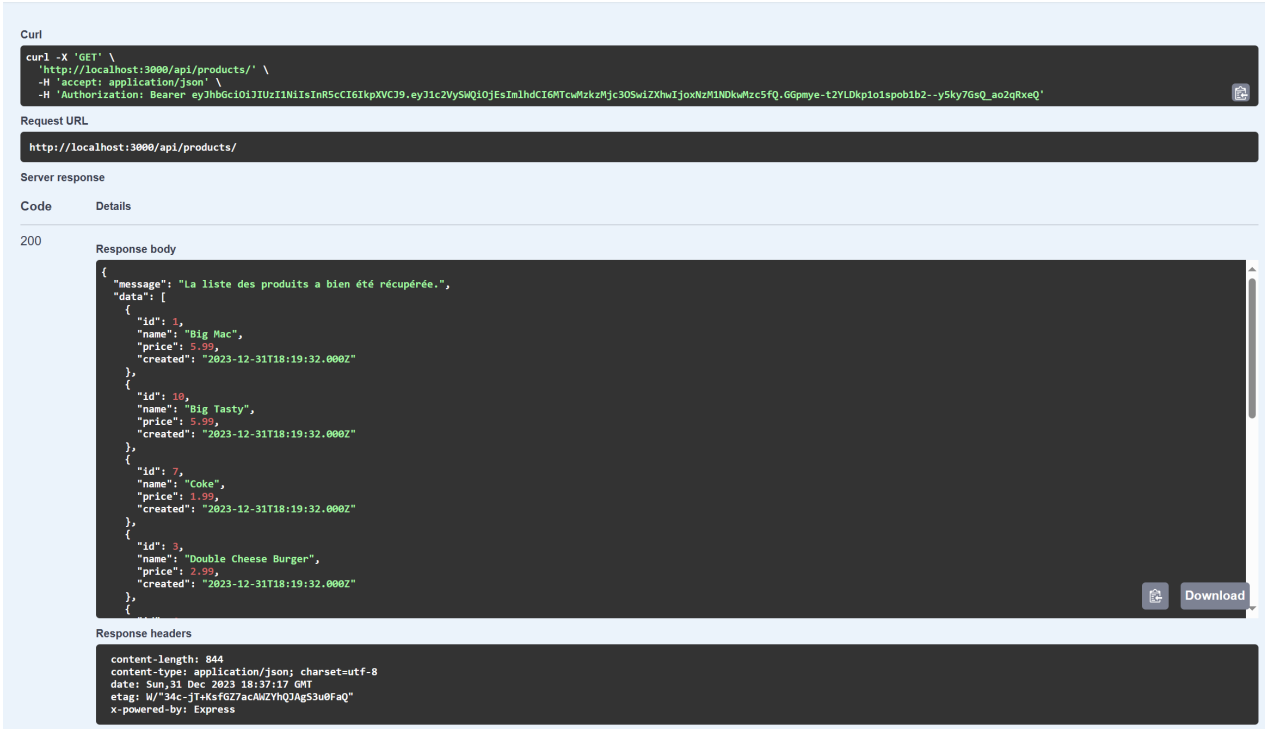
Puis copier / coller le jeton JWT :



Puis cliquer sur le bouton `Authorize`.

Et enfin le bouton `Close` :

Et maintenant l'appel HTTP à notre API REST depuis la documentation fonctionne



Voila ! Il ne reste plus qu'à documenter les autres routes !

Je vous laisse cela comme exercice :-)