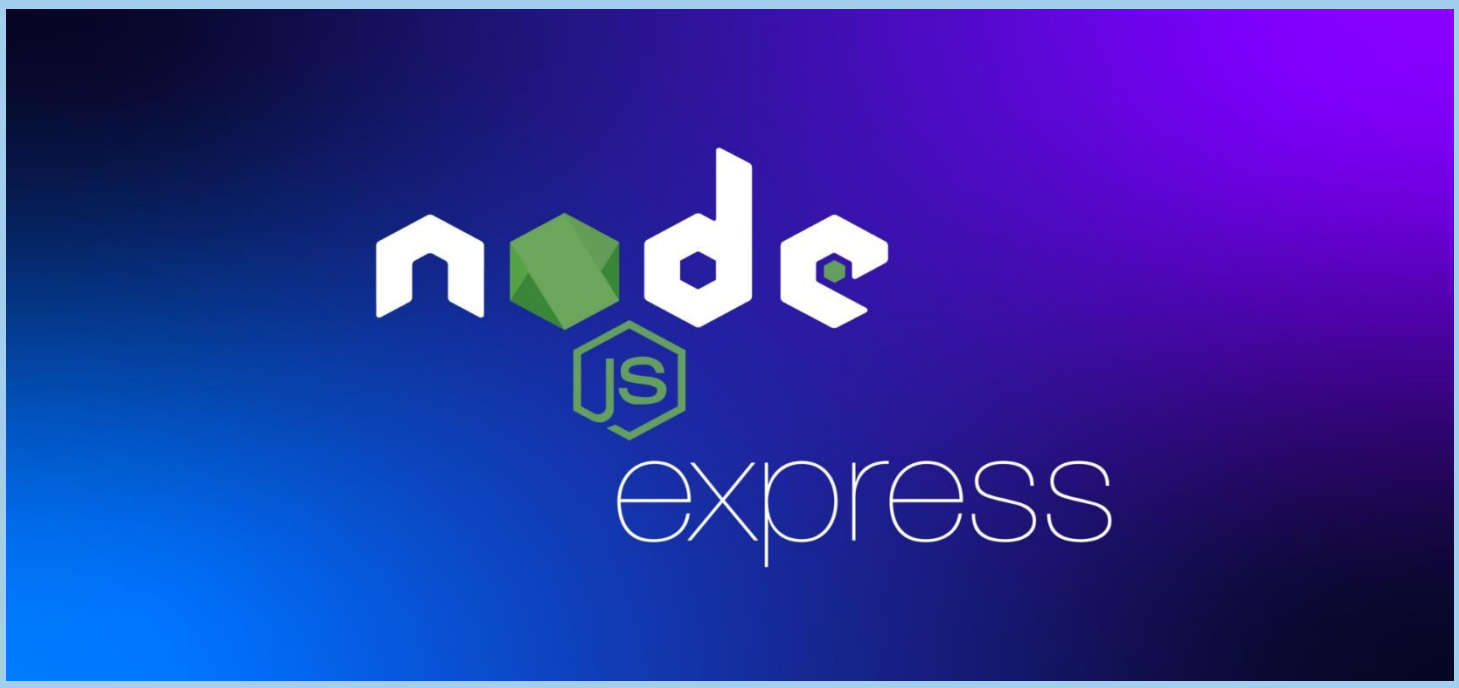


Manuel De Formation
NodeJS & Express JS

Développez des applications performantes et évolutives avec Node.js et Express.js

Manuel de formation NodeJS et ExpressJS

Développez des applications performantes et évolutives
avec Node.js et Express.js

Département : INNOVATON

Responsable : Wabenga Alphonse

Responsable adjoint : Dukizwe Darcy

Première publication : Mai 2024

Dernière modification : Mai 2024

MEDIABOX

Table de matière

Introduction

Le contenu de ce guide vise non pas à enseigner exhaustivement chaque technologie présentée, mais à fournir des directives et des bonnes pratiques pour l'utilisation de ces outils. Pour chacune des technologies ou outils mentionnés dans ce guide, vous trouverez un lien renvoyant vers une description complète pour approfondir vos connaissances sur la mise en œuvre de cet outil.

Express.js peut être utilisé pour créer différents types d'applications. Cependant, le but de ce document est de vous fournir des directives pour créer une API REST que vous pourrez ensuite consommer à travers une application.

A qui appartient ce guide

Ce manuel s'adresse aux développeurs de la société Mediabox, visant à approfondir et enrichir leurs compétences dans l'utilisation de Node.js et Express.js pour le développement d'API. Il est expressément interdit de diffuser le contenu de ce guide à des individus n'appartenant pas à l'entreprise, dans le but de garantir une intégrité optimale.

Objectifs de la Formation

À l'issue de cette formation, l'apprenant sera en mesure de : - Mettre en place un nouveau projet grâce à npm - Maîtriser les bases de Node.js et d'Express.js - Utiliser efficacement un ORM Sequelize - Structurer de manière optimale un projet - Mettre en place l'internationalisation d'une application - Appliquer les bonnes pratiques pour développer une API performante et sécurisée

Liste des technologies et outils

Ceci est une liste des principales technologies et outils présentés dans ce guide :

- NodeJs - Permet d'exécuter du code JavaScript côté serveur
- NPM (Node Package Manager) - Est le gestionnaire de paquets officiel pour l'écosystème Node.js
- ExpresJS - Un framework web pour Node.js qui simplifie la création d'une application
- Sequelize - Un ORM (Object-Relational Mapping) pour Node.js qui simplifie l'interaction avec une base de données relationnelle
- JWT (JSON Web Token) - Un format ouvert utilisé pour transmettre des assertions entre un serveur et un client. Il est couramment utilisé dans le contexte de l'authentification et de l'autorisation dans les applications web et les services.
- Bcrypt - Une bibliothèque Node.js qui offre des fonctions de hachage de mot de passe sécurisées
- Express-fileupload - Un middleware pour Express.js qui facilite la gestion des téléchargements de fichiers côté serveur dans une application Express.
- Dotenv - Une bibliothèque Node.js qui facilite la gestion des variables d'environnement dans les applications
- Nodemon - Un utilitaire qui surveille toute modification dans votre code source et redémarre automatiquement votre serveur. Parfait pour le développement.

Prérequis

Pour tirer pleinement profit de cette formation, assurez-vous de disposer des compétences préalables suivantes :

- Des bases en Javascript (avoir déjà vu les conditions, variable et boucles aidera à la compréhension)
- Comprenez le concept d'asynchronisme en JavaScript (Voir plus)
- Familiarité avec l'utilisation du terminal.
- Comprenez les bases des systèmes de gestion de bases de données (SQL)
- Savoir utiliser un éditeur de code (on utilisera ici Visual Studio Code)

Déroulement de la formation

La formation sera divisée en plusieurs sections. La première partie abordera les fondamentaux de la technologie Node.js avec Express.js. Dans la deuxième partie, nous dresserons la liste des meilleures bibliothèques à utiliser pour développer une application performante et sécurisée. Enfin, nous conclurons par un exemple pratique visant à mettre en application les connaissances acquises.

Préparation d'un environnement

Pour créer un nouveau projet avec Node.js et Express.js:

- Assurez-vous d'avoir Node.js installé sur votre machine. Vous pouvez télécharger la dernière version depuis le site officiel de Node.js : https://nodejs.org/.
- npm (Node Package Manager) est généralement inclus avec l'installation de Node.js.

Pour vérifier si Node.js et npm (Node Package Manager) sont installés sur votre machine, vous pouvez utiliser le terminal ou l'invite de commandes. Voici comment procéder en fonction de votre système d'exploitation :

Node.js

```
node -v
```

npm

```
npm -v
```

Si Node.js et npm sont installés, les commandes ci-dessus afficheront les versions correspondantes. Si Node.js oui npm ne sont pas installés, vous obtiendrez généralement un message indiquant que la commande n'est pas reconnue.

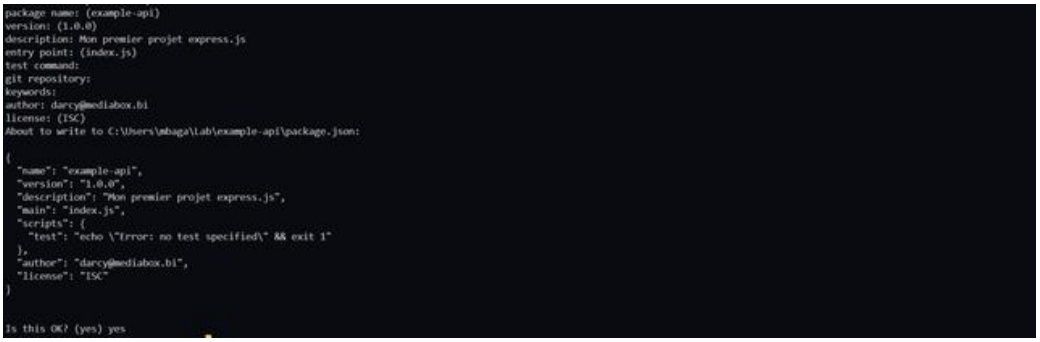
Création d'un nouveau projet

Initialisation d'un nouveau projet

Pour créer un nouveau projet, commencez par créer un dossier qui servira de répertoire principal pour votre projet. Ensuite, ouvrez votre terminal ou invite de commandes, déplacez-vous vers l'emplacement où vous avez créé le projet, et exécutez la commande suivante :

```
npm init
```

Cette commande vous posera quelques questions sur votre projet (par exemple, le nom, la version, la description, le point d'entrée, les tests, le référentiel, les mots-clés, l'auteur, la licence, etc.). Vous pouvez appuyer sur "Enter" pour accepter les valeurs par défaut ou saisir vos propres informations.



Init new project

Après avoir répondu à ces questions (ou accepté les valeurs par défaut), deux fichiers: `package.json` et `package-lock.json` seront créés dans le répertoire de votre projet, contenant les informations que vous avez fournies. Ces fichier sont essentiels pour la gestion des dépendances, la configuration du projet, et d'autres aspects liés au développement Node.js.

Installation d'une bibliothèque

Pour installer une bibliotheque, la premiere etape est de chercher le nom de la bibliotheque qui correpond a vos besoins, une fois que vous avez identifié une bibliothèque qui correspond à vos besoins, cliquez sur le lien vers la page de la bibliothèque sur le site npm. Là, vous trouverez des informations sur la bibliothèque, y compris sa documentation.

Retournez dans votre projet et exécutez la commande `npm install` suivie du nom de la bibliothèque. Par exemple, si vous avez choisi la bibliothèque "date-fns", vous pouvez exécuter :

```
npm install date-fns
```

Après l'installation, npm met à jour le fichier `package.json` avec les informations de la bibliothèque installée. Vous verrez une nouvelle entrée dans la section des dépendances.

Express

Installation d'express.js

Utilisez la commande suivante pour installer Express.js dans votre projet :

```
npm install express
```

Fichier d'entré de l'application

Lorsque vous créez une application Express.js, vous devez spécifier un fichier qui sera l'entrée principale de votre application. Ce fichier est généralement appelé "point d'entrée" ou "fichier d'application". Par convention, il est souvent nommé `app.js` ou `index.js`.

Dans ce fichier, vous configurez votre application Express en définissant des routes, des middlewares, et en écoutant les requêtes sur un port spécifique.

Créez un fichier d'application `index.js` dans votre projet et configurez-le pour utiliser Express.

Voici un exemple de contenu pour un fichier d'entrée d'application en Express.js :

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send('Bienvenue sur votre application Express.js !');
});

app.listen(port, () => {
  console.log(`Serveur écoutant sur le port ${port}`);
});
```

Voici une explication détaillée de ce que fait chaque partie du code:

1. Importation du module Express

```
// Le code importe le module Express.js et le stocke dans la variable express.
const express = require('express');
```

2. Création d'une instance de l'application Express

```
// Une instance de l'application Express est créée en appelant la fonction express(). Cette instance représente votre application web.
const app = express();
```

3. Définition d'une route pour la racine ('/')

```
// Une route est définie pour la méthode HTTP GET sur la racine de l'application ('/')
app.get('/', (req, res) => {
  res.send('Bienvenue sur votre application Express.js !');
});
```

4. Configuration du port d'écoute

```
// Le numéro de port sur lequel le serveur écoutera les requêtes est spécifié. Dans cet exemple, le port 3000 est utilisé.
const port = 3000;
```

5. Lancement du serveur :

```
// Le serveur est démarré et configuré pour écouter les connexions entrantes sur le port spécifié (3000 dans cet exemple). Lorsque le serveur est prêt à recevoir des connexions, la fonction de rappel est exécutée, affichant le message "Serveur écoutant sur le port 3000" dans la console.
app.listen(port, () => {
  console.log(`Serveur écoutant sur le port ${port}`);
});
```

Démarrage de l'Application

Exécutez votre application avec la commande suivante dans le terminal :

```
node index.js
```

Apres avoir executer cette commande, vous verrez dans le terminal le message `Serveur écoutant sur le port 3000` que nous avons configuré une fois l'application démarrée.

Test de l'Application

Pour tester une application API, vous pouvez utiliser des outils tels que Postman ou Insomnia pour envoyer des requêtes HTTP à votre API et vérifier les réponses.

Pour utiliser ces outils, rendez-vous sur leur site officiel, téléchargez le logiciel, puis procédez à l'installation sur votre ordinateur. Une fois que le logiciel est en cours d'exécution, créez une nouvelle requête en précisant l'URL de votre API pour inspecter la réponse.

Pour la suite de cette formation, nous utiliserons Thunder Client, une extension simple et populaire pour VSCode. L'avantage est que vous n'aurez pas besoin d'installer d'autres logiciels, car l'extension sera directement intégrée à votre éditeur de texte.

Voici les étapes pour installer Thunder Client dans VSCode:

1. Commencez par installer l'extension en passant par ici
2. Une fois installée, vous remarquerez une icône Thunder Client dans le sidebar de VSCode.
3. Cliquez sur l'icône Thunder Client, puis sélectionnez **`New Request`** pour tester une nouvelle requête.
4. Un nouvel onglet s'ouvrira, vous permettant de spécifier le lien de la requête à tester. Choisissez la méthode **`GET`** et saisissez l'URL `http://localhost:3000` dans la barre d'adresse, puis cliquez sur le bouton **`SEND`** comme illustré dans l'image ci-dessous :

5. Une fois la requête soumise, vous verrez le message `Bienvenue sur votre application Express.js !`` que nous avons renvoyé en réponse à la requête sur la route racine.

Bravo ! Vous venez de créer votre première application Express.js !

Structure

Bien structurer un projet Node.js avec Express offre plusieurs avantages, tels que la facilité de maintenance, la scalabilité, la lisibilité du code, et la facilité d'ajout de nouvelles fonctionnalités. Voici à quoi pourrait ressembler une bonne structure pour un projet Node.js avec Express :

```
- class/
  ├─ uploads/
  │  └─ UsersUpload.js
  │  └─ AdminUpload.js
  ├─ Upload.js
  ├─ Validation.js
- config/
  ├─ lang/
  │  └─ en.json
  │  └─ fr.json
  ├─ app.js
- constants/
  ├─ RESPONSE_CODES.js
  ├─ RESPONSE_STATUS.js
- controllers/
  ├─ auth/
  ├─ admin/
  ├─ service/
- crons/
  ├─ SENDING_PROMOTIONS_EMAILS.js
- keys/
  ├─ firebase.json
- middlewares/
  ├─ bindUser.js
  ├─ requireAuth.js.js
- models/
  ├─ User.js
- node_modules/
- public/
- routes/
  ├─ auth
  ├─ admin
  ├─ service
- socket/
  ├─ events.js
  ├─ index.js
- utils/
  ├─ sequerize.js
  ├─ randomInt.js
- views/
  ├─ emails
.env
.gitignore
package-lock.json
package.json
server.js
```

Voici une description détaillée de chaque répertoire et fichier :

- `class/`: Contient des classes qui sont utilisées dans l'application.
- `config/`: Contient des fichiers de configuration de l'application, notamment pour la gestion des langues.
- `constants/`: Dossier contenant des fichiers définissant des constantes pour l'application.
- `controllers/`: Dossier qui peut contenir des contrôleurs, des fichiers qui gèrent la logique de contrôle de votre application.
- `crons/`: Dossier qui peut contenir des fichiers liés à l'exécution de tâches cron, par exemple, pour l'automatisation de certaines actions.
- `keys/`: Dossier qui pourrait contenir des clés, par exemple, des fichiers de configuration ou des clés d'API spécifiques à votre application.
- `middlewares/`: Dossier pour les middlewares, des fonctions intermédiaires qui peuvent être utilisées pour intercepter ou modifier des requêtes HTTP.
- `models/`: Dossier qui contient les définitions de modèles de données pour l'application.
- `node_modules/`: Dossier où les dépendances de votre projet sont installées par npm.
- `public/`: Dossier où vous placez des fichiers statiques tels que des images, des styles CSS et des scripts JavaScript, qui seront accessibles publiquement.
- `routes/`: Dossier qui peut contenir les fichiers de définition des routes pour votre application.
- `socket/`: Dossier qui peut contenir des fichiers liés à la gestion des sockets, utilisés pour la communication en temps réel.
- `utils/`: Dossier qui contient des utilitaires, par exemple, des fonctions ou des configurations réutilisables.
- `views/`: Dossier qui contient des fichiers de vue si votre application utilise un moteur de template.
- `.env`: Fichier de configuration pour les variables d'environnement.
- `.gitignore`: Fichier spécifiant les fichiers et répertoires à ignorer lors de la gestion de version avec Git.
- `package-lock.json`: Fichier généré par npm pour fixer les versions exactes des dépendances.
- `package.json`: Fichier de configuration de Node.js listant les métadonnées du projet et les dépendances.
- `server.js`: Fichier principal où le serveur Express est configuré et démarré.

Comprendre express.js

Les routes

Introduction

Les routes sont un aspect important pour définir le comportement d'une application. Les routes déterminent comment l'application réagit aux requêtes HTTP des clients, en fonction de l'URL demandée et de la méthode HTTP utilisée (GET, POST, etc.).

Voici quelques-unes des méthodes HTTP les plus couramment utilisées :

Méthode	Description
GET	Récupérer des données à partir du serveur.
POST	Soumettre des données au serveur pour traitement ou enregistrement.
PUT	Mettre à jour des données sur le serveur.
DELETE	Supprimer des données sur le serveur.

Trouve ici une liste complète des méthodes HTTP

Création d'une route

Pour créer une route, Créez un fichier nommé `utilisateurs.routes.js` dans le dossier `routes`. Ce fichier sera dédié à la définition des routes spécifiques aux utilisateurs. Veuillez inclure le code suivant :

```
// utilisateurs.routes.js
const express = require("express")
const utilisateurs_routes = express.Router("")
const utilisateurs_controller = require("../controllers/utilisateurs.controller")

utilisateurs_routes.get("/utilisateurs", utilisateurs_controller.getUtilisateurs)

module.exports = utilisateurs_routes
```

Ce code spécifie une seule route GET `"/utilisateurs"` qui sera gérée par la fonction `getUtilisateurs` du contrôleur des utilisateurs.

Les controllers

Maintenant apres avoir creer une route, creer un autre fichier `utilisateurs.controller.js` dans le dossier `controllers` qui va contenier les fonctions qui vont agir comme handler pour les routes des utilisteurs:

```
// utilisateurs.controller.js
const getUtilisateurs = (req, res) => {
     try {
          const utilisateurs = [{
               id: 1,
               nom: "Jean",
               prenom: "Paul"
          }, {
               id: 2
               nom: "Marie",
               prenom: "Rose"
          }]
          res.status(200).json(utilisateurs)
     } catch (error) {
          res.status(500).send("Erreur interne du serveur")
     }
}
module.exports = {
     getUtilisateurs
}
```

Ce code définit une fonction asynchrone qui simule la récupération d'utilisateurs (à partir d'une base de données ou d'une source de données externe). En cas de succès, elle renvoie une réponse JSON contenant un tableau d'utilisateurs, et en cas d'échec, elle renvoie une réponse avec un statut HTTP 500 et un message d'erreur générique.

N'oubliez pas d'exporter les fonctions avec `module.exports` afin qu'elles puissent être utilisées ailleurs dans l'application. Voir les modules

Voici une liste de quelques codes de statut HTTP (codes de réponse) couramment utilisés:

Code	Statut	Description
200	OK	La requête a réussi. C'est le code de statut standard pour les réponses réussies.
201	CREATED	La requête a été correctement traitée, et une nouvelle ressource a été créée en conséquence.
401	UNAUTHORIZED	L'accès à la ressource est refusé en raison d'absence d'authentification ou d'informations d'identification invalides.
403	FORBIDDEN	L'accès à la ressource est refusé pour des raisons autres que l'authentification.
404	NOT FOUND	La ressource demandée n'a pas été trouvée sur le serveur.
500	INTERNAL SERVER ERROR	Erreur générique indiquant qu'une condition inattendue a empêché le serveur de satisfaire la requête.

Liste complète des codes de statut HTTP

Liaison du point d'entrée et les routes

Maintenant que nous avons créé nos routes et défini les fonctions qui seront appelées lorsqu'elles seront sollicitées, nous allons les configurer dans le fichier `server.js` afin de les rendre identifiables au sein de l'application.

Ainsi, le contenu du fichier `server.js` ressemblera à ceci :

```
// server.js
const express = require('express');
const utilisateurs_routes = require('./routes/utilisateurs.routes');
const app = express();
const port = 3000;

app.use('/', utilisateurs_routes)

app.listen(port, () => {
  console.log(`Serveur écoutant sur le port ${port}`);
});
```

Dans ce code, `app.use('/', utilisateurs_routes)` montre que les routes définies dans `utilisateurs.routes.js` sont montées sur la racine de l'application. `'/'` spécifie le chemin de base pour lequel le middleware sera activé. Nous aborderons les middlewares un peu plus tard.

Tester l'application

Maintenant que tout est configuré, nous pouvons procéder aux tests de notre route. Pour ce faire, démarrez l'application en utilisant la commande suivante :

```
node server.js
```

Une fois l'application démarrée, ouvrez l'extension `Thunder Client`, choisissez la méthode `GET`, puis saisissez l'adresse `http://localhost:3000/utilisateurs`. Vous verrez ainsi une liste des utilisateurs dans la réponse, conformément à ce que nous avons défini dans notre fonction.

Les middlewares

Les middlewares sont des fonctions qui ont accès à l'objet de requête (req), à l'objet de réponse (res), et à la fonction next dans le cycle de vie de la requête. Ils sont utilisés pour effectuer des opérations telles que la modification des objets de requête et de réponse

On distingue trois types de middlewares :

1. `Middlewares intégrés` : Express offre des middlewares intégrés pour des tâches courantes comme le traitement des données JSON, des formulaires, la gestion des cookies, etc.
2. `Middleware Global` : qui sont exécutés pour chaque requête, indépendamment de la route spécifique.
3. `Les middlewares de route` : qui sont spécifiés pour une route particulière et sont exécutés uniquement lorsque cette route est atteinte.

```
const express = require('express');
const utilisateurs_routes = require('./routes/utilisateurs.routes');
const app = express();
const port = 3000;

// Middleware intégré pour le traitement des données JSON
app.use(express.json());

// Middleware intégré pour le traitement des formulaires HTML
app.use(express.urlencoded({ extended: true }));

// Middleware global
app.use(monMiddleware);

// Middleware spécifique à une route
app.use('/', utilisateurs_routes)

app.listen(port, () => {
  console.log(`Serveur écoutant sur le port ${port}`);
});
```

Les middlewares jouent un rôle essentiel dans l'authentification pour vérifier si un utilisateur est correctement authentifié avant de permettre l'accès à certaines routes ou ressources.

Comprendre la requête et la réponse

Dans Express.js, “req” et “res” sont des abréviations couramment utilisées pour représenter les objets de requête (request) et de réponse (response) dans le contexte d’une application web. Ces objets sont essentiels pour gérer les requêtes HTTP entrantes et générer les réponses correspondantes.

La requête

`req (Request)` : Cet objet représente la requête HTTP entrante et contient des informations sur la requête telle que les paramètres de l’URL, les données du corps de la requête(body), les en-têtes(headers), etc. Vous pouvez accéder à ces informations à l’intérieur de vos routes pour prendre des décisions en fonction de la requête.

Exemple d’utilisation dans une route :

```
app.get('/exemple', (req, res) => {
  // Accéder aux paramètres de l'URL
  const id = req.params.id;

  // Accéder aux données du corps de la requête (si c'est une requête POST)
  const data = req.body;

  // Autres informations disponibles dans l'objet req
  const headers = req.headers;
  const method = req.method;

  // Logique de traitement de la requête ici
});
```

La réponse

`res (Response)` : Cet objet représente la réponse HTTP que votre serveur génère et envoie en réponse à la requête. À travers l’objet `res`, vous pouvez définir des en-têtes de réponse, envoyer des données au client, rediriger vers d’autres URL, etc.

Exemple d’utilisation dans une route :

```
app.get('/exemple', (req, res) => {
  // Envoi d'une réponse simple au client
  res.send('Bonjour, monde!');

  // Envoi de données JSON
  res.json({ message: 'Ceci est un exemple JSON' });

  // Définition d'un code d'état de la réponse
  res.status(200); // OK
});
```

Nodemon

nodemon est un utilitaire de surveillance pour les applications Node.js. Il permet de redémarrer automatiquement le serveur Node.js chaque fois qu'un changement est détecté dans les fichiers source de l'application. Cela facilite le processus de développement, car vous n'avez pas besoin de redémarrer manuellement le serveur à chaque modification de code. Pour utiliser nodemon, vous devez l'installer globalement ou localement dans votre projet, selon vos préférences. Vous pouvez l'installer à l'aide de la commande npm :

```
npm install -g nodemon
```

Ensuite, vous pouvez exécuter votre application Node.js avec nodemon au lieu de node. Par exemple:

```
nodemon server.js
```

Pour simplifier les choses, vous pouvez définir un script `start` dans le fichier `package.json` comme suit:

```
// pockage.json
{
  "name": "example-api",
  "version": "1.0.0",
  "description": "Mon premier projet express.js",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "nodemon server.js"
  },
  "author": "darcy@mediabox.bi",
  "license": "ISC",
  "dependencies": {
    "dotenv": "^16.3.2",
    "express": "^4.18.2",
    "mysql2": "^3.7.1",
    "sequelize": "^6.35.2",
    "sqlite3": "^5.1.7"
  }
}
```

Maintenant pour démarrer l'application, il suffit d'exécuter la commande :

```
npm run start
```

ou tout simplement

```
npm start
```

Fichier d'environnement

Un fichier `.env` est souvent utilisé pour stocker des variables d'environnement. Les variables d'environnement sont des valeurs sensibles ou configurables que vous ne souhaitez pas hardcoder dans votre code source.

Pour charger les variables d'environnement à partir du fichier `.env`, vous devez installer le module dotenv. Vous pouvez le faire en exécutant la commande suivante dans votre terminal :

```
npm install dotenv
```

Créez un fichier appelé `.env` à la racine de votre projet. Ce fichier contiendra vos variables d'environnement, par exemple :

```
PORT = 3000
BD_HOST = localhost
DB_USER = root
DB_PASSWORD =
DB_NAME = formation_node
```

Au début du fichier où vous souhaitez charger les variables (par exemple, app.js ou autre), ajoutez les lignes suivantes pour charger les variables d'environnement à partir du fichier `.env`:

```
const dotenv = require('dotenv')
dotenv.config()
```

Par exemple, pour charger le port de l'application à partir du fichier `.env`, le point d'entrée serait le suivant :

```
// server.js
const express = require('express');
const utilisateurs_routes = require('./routes/utilisateurs.routes');
const app = express();
const dotenv = require('dotenv')
dotenv.config()

const port = process.env.PORT;

app.use(express.json());
app.use(express.urlencoded({ extended: true }));

// Middleware spécifique à une route
app.use('/', utilisateurs_routes)

app.listen(port, () => {
  console.log(`Serveur écoutant sur le port ${port}`);
});
```

Interagir avec une base de données

Commençons par créer une table `utilisateurs` qui nous servira d'exemple. La table comporte trois champs : ID_UTILISATEUR, NOM et PRENOM.

```
CREATE TABLE `utilisateurs` (
 `ID_UTILISATEUR` int(11) NOT NULL AUTO_INCREMENT,
 `NOM` varchar(50) NOT NULL,
 `PRENOM` varchar(50) NOT NULL,
 PRIMARY KEY (`ID_UTILISATEUR`)
)
```

Requête préparée

L'avantage principal des requêtes préparées réside dans leur capacité à prévenir les attaques par injection SQL. En isolant les valeurs des paramètres de la requête SQL, même si ces valeurs sont fournies par l'utilisateur, elles ne peuvent pas altérer la structure de la requête.

Prenons par exemple une situation où l'on doit effectuer une recherche d'un nom dans la table `utilisateurs`, une opération initiée par une saisie fournie par un utilisateur de l'application. La requête serait alors formulée comme suit :

```
// utilisateurs.controller.js
const { query } = require("../utils/db")
const getUtilisateurs = async (req, res) => {
     try {
          const { nom } = req.query
          const utilisateurs = await query("SELECT ID_UTILISATEUR, NOM, PRENOM FROM u
tilisateurs WHERE NOM = ?", [nom])
          res.status(200).json(utilisateurs)
     } catch (error) {
          res.status(500).send("Erreur interne du serveur")
     }
}
module.exports = {
     getUtilisateurs
}
```

Sequelize

Sequelize permet de simplifier l'interaction avec les bases de données relationnelles dans des applications Node.js. Elle est principalement utilisée comme ORM (Object-Relational Mapping) pour des bases de données SQL qui facilite la manipulation des données dans la base de données en utilisant des objets JavaScript plutôt que des requêtes SQL directes.

Installation et configuration

Pour commencer à utiliser Sequelize, vous devez d'abord l'installer dans votre projet en utilisant la commande suivante :

```
npm install sequelize sqlite3
```

Pour simplifier son utilisation, nous allons également créer un fichier `sequelize.js` dans le dossier `utils` et y insérer le code suivant :

```
// utils/sequelize.js
const { Sequelize } = require('sequelize');
const dotenv = require('dotenv')

dotenv.config()
const sequelize = new Sequelize({
     host: process.env.BD_HOST,
     username: process.env.DB_USER,
     password: process.env.DB_PASSWORD,
     database: process.env.DB_NAME,
     port: process.env.DB_PORT ? process.env.DB_PORT : 3306,
     dialect: 'mysql'
})
module.exports = sequelize
```

Les modèles

Les modèles sont des représentations JavaScript des tables dans une base de données relationnelle. Ils sont utilisés pour définir la structure des données et les relations entre les tables.

Dans notre cas, pour créer un modèle pour la table `utilisateurs`, créez un fichier `Utilisateurs.js` dans le dossier models et insérez-y le code suivant :

```
// models/Utilisateurs.js
const { DataTypes } = require('sequelize');
const sequelize = require('../utils/sequerize');

const Utilisateurs = sequelize.define('utilisateurs', {
     ID_UTILISATEUR: {
          type: DataTypes.INTEGER,
          allowNull: false,
          primaryKey: true,
          autoIncrement: true
     },
     NOM: {
          type: DataTypes.STRING(50),
          allowNull: false
     },
     PRENOM: {
          type: DataTypes.STRING(50),
          allowNull: false
     }
}, {
     freezeTableName: true,
     tableName: 'utilisateurs',
     timestamps: false
})

module.exports = Utilisateurs
```

Convention : les noms de fichiers des modèles commencent toujours par une lettre majuscule.

Maintenant que notre modèle est créé, nous pouvons l'utiliser de la manière suivante pour récupérer la liste des utilisateurs :

```
const Utilisateurs = require("../models/Utilisateurs")

const getUtilisateurs = async (req, res) => {
     try {
          const utilisateurs = await Utilisateurs.findAll()
          res.status(200).json(utilisateurs)
     } catch (error) {
          res.status(500).send("Erreur interne du serveur")
     }
}
module.exports = {
     getUtilisateurs
}
```

Un CRUD avec sequelize

Maintenant que nous avons vu les bases de l'utilisation de Sequelize, réalisons un petit CRUD de la table `utilisateurs`.

Commençons par ajouter dans le fichier `utilisateurs.routes.js` les routes de création, de modification et de suppression d'un utilisateur.

```
// routes/utilisateurs.routes.js
const express = require("express")
const utilisateurs_routes = express.Router("")
const utilisateurs_controller = require("../controllers/utilisateurs.controller")

utilisateurs_routes.post("/utilisateurs", utilisateurs_controller.creerUtilisateur)
utilisateurs_routes.get("/utilisateurs", utilisateurs_controller.getUtilisateurs)
utilisateurs_routes.get("/utilisateurs/:ID_UTILISATEUR", utilisateurs_controller.findBy
id)
utilisateurs_routes.put("/utilisateurs/:ID_UTILISATEUR", utilisateurs_controller.modifi
erUtilisateur)
utilisateurs_routes.delete("/utilisateurs/:ID_UTILISATEUR", utilisateurs_controller.sup
primerUtilisateur)

module.exports = utilisateurs_routes
```

Nous avons ajouter le paramètre `:ID_UTILISATEUR` sur les routes de modification et de suppression d'un utilisateur. Cela indique que nous devons récupérer l'ID de l'utilisateur fourni en dernier dans ces routes. En procédant ainsi, nous aurons la possibilité de récupérer ces IDs dans la requête avec `req.params`.

Dans le contrôleur, nous allons passer à la création des fonctions pour chaque route:

```
// controllers/utilisateurs.controller.js
const Utilisateurs = require("../models/Utilisateurs")
const getUtilisateurs = async (req, res) => {
     try {
          const utilisateurs = await Utilisateurs.findAll()
          res.status(200).json(utilisateurs)
     } catch (error) {
          res.status(500).send("Erreur interne du serveur")
     }
}
const findByid = async (req, res) => {
     try {
          const { ID_UTILISATEUR } = req.params
          const utilisateur = await Utilisateurs.findOne({
               where: {
                    ID_UTILISATEUR: ID_UTILISATEUR
               }
          })
          res.status(200).json(utilisateur)
     } catch (error) {
          console.log(error)
```

```
            res.status(500).send("Erreur interne du serveur")
        }
}

const creerUtilisateur = async (req, res) => {
        try {
            const { NOM, PRENOM } = req.body
            const nouveauUtilisateur = await Utilisateurs.create({
                NOM: NOM,
                PRENOM: PRENOM
            })
            res.status(200).json({
                message: "Nouvel utilisateur créé avec succès",
                data: nouveauUtilisateur
            })
        } catch (error) {
            console.log(error)
            res.status(500).send("Erreur interne du serveur")
        }
}

const modifierUtilisateur = async (req, res) => {
        try {
            const { ID_UTILISATEUR } = req.params
            const { NOM, PRENOM } = req.body
            await Utilisateurs.update({
                NOM: NOM,
                PRENOM: PRENOM
            }, {
                where: {
                    ID_UTILISATEUR: ID_UTILISATEUR
                }
            })
            res.status(200).json({
                message: "Utilisateur modifié avec succès",
                data: {
                    ID_UTILISATEUR,
                    NOM,
                    PRENOM
                }
            })
        } catch (error) {
            console.log(error)
            res.status(500).send("Erreur interne du serveur")
        }
}

const supprimerUtilisateur = async (req, res) => {
        try {
            const { ID_UTILISATEUR } = req.params
            await Utilisateurs.destroy({
```

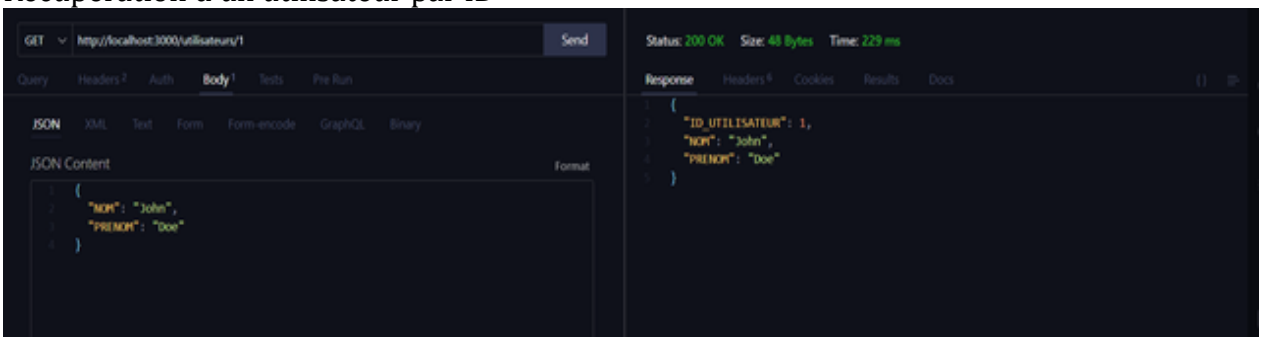
```
                where: {
                    ID_UTILISATEUR: ID_UTILISATEUR
                }
            })
            res.status(200).json({
                message: "L'utilisateur a été supprimé avec succès"
            })
        } catch (error) {
            console.log(error)
            res.status(500).send("Erreur interne du serveur")
        }
}

module.exports = {
    getUtilisateurs,
    creerUtilisateur,
    modifierUtilisateur,
    supprimerUtilisateur
}
```

Maintenant que le code est prêt, nous pouvons tester les différentes routes avec `Thunder Client`:

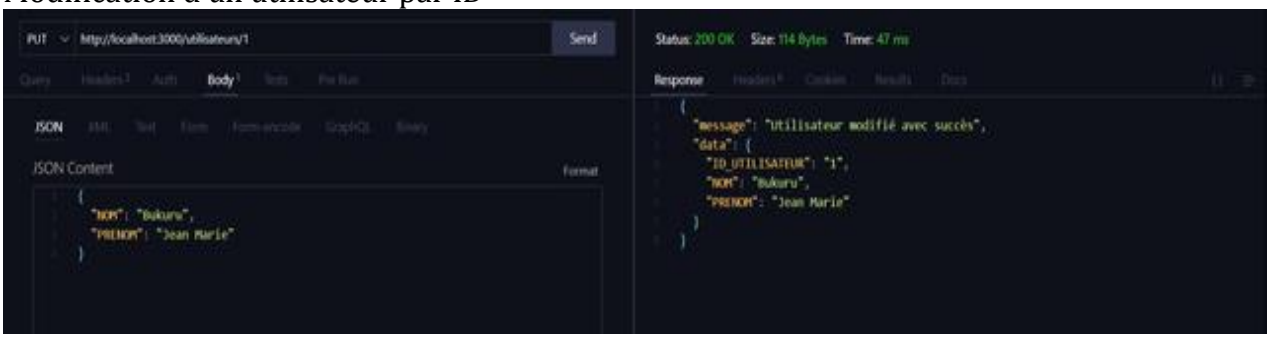
Insertion d'un utilisateur

Récupération de la liste utilisateurs

Récupération d'un utilisateur par ID



Modification d'un utilisateur par ID



Suppression d'un utilisateur par ID

Les associations

Les associations en Sequelize sont des liens entre différents modèles (tables) dans une base de données relationnelle. Ces associations définissent comment les données de différents modèles sont connectées les unes aux autres. Sequelize offre plusieurs types d'associations, notamment :

- `BelongsTo`: Une association "belongsTo" est utilisée pour indiquer qu'une instance d'un modèle appartient à une autre instance d'un modèle. Par exemple, si vous avez une table "Commentaire" et une table "Utilisateur", vous pourriez définir une association "belongsTo" pour indiquer qu'un commentaire appartient à un utilisateur.

- `HasOne`: Une association "hasOne" est utilisée pour indiquer qu'une instance d'un modèle a exactement une autre instance d'un autre modèle associée. Cela est utile lorsque la relation est de type un à un.

- `HasMany`: Une association "hasMany" est utilisée pour indiquer qu'une instance d'un modèle peut avoir plusieurs instances d'un autre modèle associé. Cela est souvent utilisé pour les relations un à plusieurs.

Revenons à l'exemple de notre table `utilisateurs` où chaque utilisateur possède un profil provenant d'une autre table appelée `profils`. Dans ce cas, nous allons créer une nouvelle table pour les profils et stocker l'ID du profil dans la table des utilisateurs en tant que clé étrangère.

```
// models/Utilisateurs.js

const { DataTypes } = require('sequelize');
const sequelize = require('../utils/sequelize');
const Profils = require('./Profils');

const Utilisateurs = sequelize.define('utilisateurs', {
     ID_UTILISATEUR: {
          type: DataTypes.INTEGER,
          allowNull: false,
          primaryKey: true,
          autoIncrement: true
     },
     NOM: {
          type: DataTypes.STRING(50),
          allowNull: false
     },
     PRENOM: {
          type: DataTypes.STRING(50),
          allowNull: false
     },
     ID_PROFIL: {
          type: DataTypes.INTEGER,
          allowNull: false
     },
}, {
     freezeTableName: true,
     tableName: 'utilisateurs',
     timestamps: false
})

Utilisateurs.belongsTo(Profils, { as: 'profil', foreignKey: "ID_PROFIL" })

module.exports = Utilisateurs
```

Comme vous pouvez le remarquer, à la fin du modèle `Utilisateurs`, nous avons ajouté une association avec l'autre modèle `Profils`.

Maintenant, lors de la récupération des enregistrements dans la table `utilisateurs`, pour obtenir le profil correspondant, Sequelize propose un paramètre `include` pour réaliser cette opération :

```
// controllers/utilisateurs.controller.js

const Profils = require("../models/Profils")
const Utilisateurs = require("../models/Utilisateurs")

const getUtilisateurs = async (req, res) => {
     try {
          const utilisateurs = await Utilisateurs.findAll({
               include: {
                    model: Profils,
                    as: 'profil'
               }
          })
          res.status(200).json(utilisateurs)
     } catch (error) {
          console.log(error)
          res.status(500).send("Erreur interne du serveur")
     }
}
```

Dans la réponse, il y aura une nouvelle clé `profil` qui contient les informations du profil.

Requête classique

Dans Node.js, il n'existe pas de moyen intégré pour interagir directement avec les bases de données, ce qui nécessite l'utilisation de bibliothèques tierces pour faciliter cette tâche.

Il existe plusieurs bibliothèques disponibles mais nous utiliserons mysql2 un pilote très populaire pour MySQL, le SGBD que nous utilisons dans cette formation.

Commencer par installer la bibliothèque en exécutant la commande suivante :

```
npm i mysql2
```

Pour simplifier son utilisation, nous allons créer un fichier `db.js` dans le dossier `utils` et y insérer le code suivant :

```
// utils/db.js
const { createPool } = require('mysql2/promise')
const dotenv = require('dotenv')

dotenv.config()
var globalPool = undefined
const connection = async () => {
     try {
          if (globalPool) return globalPool
          globalPool = await createPool({
               host: process.env.BD_HOST,
               user: process.env.DB_USER,
               password: process.env.DB_PASSWORD,
               database: process.env.DB_NAME,
               port: process.env.DB_PORT ? process.env.DB_PORT : 3306
          })
          return globalPool
     } catch (error) {
          throw error
     }
}
```

```
const query = async (query, values) => {
     const pool = await connection()
     return (await pool.query(query, values))[0]
}

module.exports = {
     connection,
     query
}
```

Ce code permet de se connecter à une base de données en utilisant les identifiants configurés dans le fichier d'environnement `.env`. Il utilise un systeme de pool pour réduire le temps nécessaire à l'établissement d'une connexion au serveur MySQL en réutilisant une connexion précédente.

Maintenant que tout est configuré, nous pouvons récupérer la liste des utilisateurs de la manière suivante :

```
// utilisateurs.controller.js
const { query } = require("../utils/db")
const getUtilisateurs = async (req, res) => {
     try {
          const utilisateurs = await query("SELECT ID_UTILISATEUR, NOM, PRENOM FROM u
tilisateurs")
          res.status(200).json(utilisateurs)
     } catch (error) {
          res.status(500).send("Erreur interne du serveur")
     }
}
module.exports = {
     getUtilisateurs
}
```

Dans ce code, nous avons d'abord importé `query` depuis le fichier `db.js` que nous avons créé. Ensuite, nous avons modifié la structure de la fonction en une fonction asynchrone (async) afin qu'elle puisse accepter l'utilisation de `await`.

Validation des données

La validation des données est une étape importante dans le développement d'applications, car elle permet de garantir que les données entrantes sont conformes aux attentes et aux règles définies. En Node.js, plusieurs bibliothèques peuvent être utilisées pour la validation des données. L'une des bibliothèques les plus populaires est Joi.

Pour la suite de cette formation, nous n'utiliserons pas "Joi". Nous disposons d'une classe appelée "Validation" que nous utiliserons pour valider les données. L'avantage de cette classe est sa simplicité et sa facilité d'utilisation, avec la possibilité d'ajouter des validations personnalisées.

Pour l'utiliser, il suffit de créer une nouvelle classe (fichier) appelée Validation.js à l'intérieur du dossier class, situé à la racine du projet, et d'y insérer le code suivant :

```
const { query } = require("../utils/db");
const moment = require("moment")

class Validation {
     constructor(data, validation, customMessages) {
          this.data = data;
          this.validation = validation;
          this.customMessages = customMessages;
          this.errors = {};
     }
     /**
      * Check if passed data are valid
      *
      * @returns Boolean
```

```
     */
    async isValidate() {
         const errors = await this.getErrors()
         return (
              Object.keys(errors).length === 0 &&
              errors.constructor === Object
         );
    }

    /**
     * set new message error
     * @param {string} key - message key
     * @param {string} message - that describe the error
     * @returns {void}
     */
    async setError(key, message) {
         const errors = this.errors[key]
              ? [...this.errors[key], message]
              : [message];
         this.errors = { ...this.errors, [key]: errors };
    }
    /**
     * get message by key
     * @param {string} key - the message key you want to get
     * @returns {string}
     */
    async getError(key) {
         await this.run();
         return this.errors[key];
    }
    /**
     * mark input data as required
     * @param {string} key - the key you want to set as required
     * @param {string} value - the value
     */
    async required(key, intitialValue) {
         try {
              if (!this.validation[key] || !this.validation[key].required) return false
              const value = typeof (intitialValue) == 'string' ? intitialValue ? intitia
lValue.trim() : '' : intitialValue
              let isInvalid = false
              if (typeof (value) == 'string' || Array.isArray(value)) {
                   if (!value || value === '' || value.length === 0) {
                        isInvalid = true
                   }
              } else if (typeof (value) == 'object' && !Array.isArray(value)) {
                   if (!value) {
                        isInvalid = true
                   }
              } else if (!value) {
                   isInvalid = true
              }
              if (isInvalid) {
```

```
                    this.setError(key, this.customMessages?.[key]?.required || `This fiel
d is required`)
               }

          } catch (error) {
               throw error
          }
     }

     async length(key, value, params) {
          if (!value) return;
          const [min, max] = params;
          if (min && !max && value.length < min) {
               this.setError(
                    key,
                    this.customMessages?.[key]?.length || `Enter et least ${min} characte
rs`
               );
          } else if (!min && max && value.length > max) {
               this.setError(
                    key,
                    this.customMessages?.[key]?.length ||
                    `You can not exceed ${max} characters`
               );
          } else if (min && max && (value.length < min || value.length > max)) {
               this.setError(
                    key,
                    this.customMessages?.[key]?.length ||
                    `The value of this filed must be between ${min} and ${max}`
               );
          }
     }
     async match(key, value, params) {
          if (!value) return;
          if (this.data[params] !== value) {
               this.setError(
                    key,
                    this.customMessages?.[key]?.match ||
                    `Value does not match the ${params} value`
               );
          }
     }
     async username(key, value) {
          if (!value) return;
          const validUsername = /^[a-zA-Z0-9._]+$/.test(value);
          if (!validUsername || value?.length < 2) {
               this.setError(
                    key,
                    this.customMessages?.[key]?.username ||
                    "Incorrect username (letters, numbers, point or underscore)"
               );
          }
     }
```

```
    async email(key, value) {
        if (!value) return;
        const validEmail = /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(value);
        if (!validEmail) {
            this.setError(
                key,
                this.customMessages?.[key]?.email || "Invalid email"
            );
        }
    }

    async image(key, value, params) {
        if (!value) return;
        const IMAGES_MIMES = ["image/jpeg", "image/jpg", "image/gif", "image/png"];
        if (!IMAGES_MIMES.includes(value.mimetype)) {
            this.setError(
                key,
                this.customMessages?.[key]?.image || "Please choose a valid image"
            );
        } else if (params < value.size) {
            const megaBite = (params - 1000000) / 1000000;
            this.setError(
                key,
                this.customMessages?.[key]?.size ||
                `Your image is too large (max: ${megaBite} MB)`
            );
        }
    }

    async fileTypes(key, value, params) {
        if (!value || !value?.mimetype) return;
        const VALID_MIMES = params
        if (!VALID_MIMES.includes(value.mimetype)) {
            this.setError(
                key,
                this.customMessages?.[key]?.fileTypes || `Invalid file type(${params.
join(', ')})`
            )
        }
    }

    async fileSize(key, value, params) {
        if (!value || !value?.size) return;
        if (params < value.size) {
            const megaBite = (params - 1000000) / 1000000;
            this.setError(
                key,
                this.customMessages?.[key]?.fileSize || `File too large (max: ${megaB
ite} MB)`
            );
        }
    }
    async exists(key, value, params) {
```

```
          try {
               if (!value) return
               const [tableName, columnName] = params.split(',')
               const row = (await query(`SELECT ${columnName} FROM ${tableName} WHERE ${c
olumnName} = ?`, [value]))[0]
               if (!row) {
                    this.setError(
                         key,
                         this.customMessages?.[key]?.exists ||
                         `This field is not defined on ${tableName} table`
                    );
               }
          } catch (error) {
               throw error
          }
     }
     async unique(key, value, params) {
          try {
               if (!value) return;
               const [tableName, columnName] = params.split(',')
               const row = (await query(`SELECT ${columnName} FROM ${tableName} WHERE ${c
olumnName} = ?`, [value]))[0]
               if (row) {
                    this.setError(
                         key,
                         this.customMessages?.[key]?.unique ||
                         `The ${columnName} must be unique on ${tableName} table`
                    );
               }
          } catch (error) {
               throw error
          }
     }
     alpha(key, value) {
          if (!value) return
          const pattern = /^[\w\s!"#$%&'()*+,-./:;<=>?@[\\\]^_`{|}~\u00C0-\u017F]+$/u
          let isString = pattern.test(value);
          if (!isString) {
               this.setError(key, this.customMessages?.[key]?.alpha || `This field must c
ontains alphanumeric characters only`)
          }
     }
     number(key, value) {
          if (!value) return
          let isnum = /^\d+$/.test(value);
          if (!isnum) {
               this.setError(key, this.customMessages?.[key]?.number || `This field must
be a valid number`)
          }
     }
     date(key, value, params) {
          if (!value) return
          const format = params
```

```
          let isDate = moment(value, format).isValid()
          if (!isDate) {
               this.setError(key, this.customMessages?.[key]?.date || `This field must be
a valid date(${format})`)
          }
     }
     /**lm!
      *
      * run the valiton to check for errors
      */
     async run() {
          for (let key in this.validation) {
               const value = this.getValue(key);
               const [properties, params] = this.getProperties(this.validation[key]);
               try {
                    await Promise.all(properties.map(async (property) => {
                         await this[property](key, value, params?.[property]);
                    }))
               } catch (error) {
                    throw error
               }
          }
     }
     /**
      * get all erros
      * @returns {object | {}}
      */
     async getErrors() {
          return this.errors;
     }

     getProperties(value) {
          switch (typeof value) {
               case "string":
                    return [value.split(","), null];

               case "object":
                    const properties = [];
                    for (let key in value) {
                         properties.push(key);
                    }
                    return [properties, value];

               default:
                    return [value, null];
          }
     }

     getValue(key) {
          return this.data && key ? this.data[key] : null
     }
}
module.exports = Validation;
```

Cette classe comprend des validations couramment utilisées, mais après l'instanciation de la classe, vous avez la possibilité de définir des validations personnalisées.

Une fois que la classe est créée, on pourra l'utiliser dans le contrôleur pour la validation des données, comme dans cet exemple :

```
// utilisateurs.controller.js
...
const Validation = require("../class/Validation")

const creerUtilisateur = async (req, res) => {
     try {
          const { NOM, PRENOM, ID_PROFIL } = req.body
          const validation = new Validation(req.body, {
               NOM: {
                    required: true,
                    alpha: true,
                    length: [2, 20]
               },
               PRENOM: {
                    required: true,
                    alpha: true,
                    length: [2, 20]
               },
               ID_PROFIL: {
                    required: true,
                    number: true,
                    exists: "profils,ID_PROFIL"
               }
          }, {
               NOM: {
                    required: "Ce champ est obligatoire",
                    alpha: "Le nom doit contenir des caractères alphanumériques",
                    length: "Le nom doit comporter entre 2 et 20 caractères"
               },
               PRENOM: {
                    required: "Ce champ est obligatoire",
                    alpha: "Le prénom doit contenir des caractères alphanumériques",
                    length: "Le prénom doit comporter entre 2 et 20 caractères"
               },
               ID_PROFIL: {
                    required: "Le profil est obligatoire",
                    number: "Ce champ doit contenir un nombre valide",
                    exists: "Le profil n'existe pas"
               }
          })
          await validation.run()
          const isValid = await validation.isValidate()
          if(!isValid) {
               const errors = await validation.getErrors()
               return res.status(422).json({
                    message: "La validation des données a echouée",
                    data: errors
```

```
            })
        }
        const nouveauUtilisateur = await Utilisateurs.create({
            NOM: NOM,
            PRENOM: PRENOM,
            ID_PROFIL: ID_PROFIL
        })
        res.status(200).json({
            message: "Nouvel utilisateur créé avec succès",
            data: nouveauUtilisateur
        })
    } catch (error) {
        console.log(error)
        res.status(500).send("Erreur interne du serveur")
    }
}
...
```

La classe `Validation` prend trois paramètres dans son constructeur.

1. `data` : c'est un objet contenant les données à valider.
2. `validation` : c'est un objet précisant les validations que vous spécifiez pour les données passées dans le premier paramètre. Les noms des clés doivent ressembler à celles qui doivent être définies dans l'objet des données à valider, et préciser la validation pour chaque clé.
3. `customMessages` : Par défaut, les messages retournés sont en anglais, mais à l'aide de ce paramètre, vous pouvez changer les messages qui seront retournés pour chaque validation échouée.

Voici une liste des validations par défaut et de leurs significations :

Validation	Exemple	Description
required	target: { required: true }	Précisez que la clé et la valeur à l'intérieur sont obligatoires
length	target: { length: [1, 2] }	Précisez la taille du champ
match	target: { match: "keyToMatch" }	Vérifier que deux valeurs sont identiques
username	target: { username: true }	Précisez qu'un champ doit avoir un nom d'utilisateur valide (lettres, chiffres, point ou underscore)
email	target: { email: true }	Précisez qu'un champ doit être un email valide

image	target: { image: 1000000 }	Précisez qu'un champ doit être une image valide. 1000000 indique la taille maximale de l'image
fileTypes	target: { fileTypes: ['application/pdf', 'image/jpeg'] }	Précisez qu'un champ doit être un fichier valide entre les types préciser dans le tableau
fileSize	target: { fileSize: 1000000 }	Précisez la taille maximale d'un fichier en octet
exists	target: { exists: "table,colonne" }	Précisez que la valeur d'un champ doit exister dans la table indiquée
unique	target: { exists: "table,colonne" }	Précisez que la valeur d'un champ doit être unique sur la table indiquée
alpha	target: { exists: true }	Précisez que la valeur d'un champ doit contenir des caractères alphanumériques
number	target: { exists: true }	Précisez que la valeur d'un champ doit être un nombre valide
date	target: { date: "DD/MM/YYYY" }	Précisez que la valeur d'un champ doit être une date au format précisé

Upload des fichiers

Introduction

Comme c'est le cas pour la gestion des bases de données, il n'existe pas de fonctionnalité intégrée dans Express.js pour faciliter le téléchargement de fichiers. Pour accomplir cette tâche, il est souvent nécessaire de recourir à des bibliothèques tierces.

Dans cette formation, nous allons explorer l'utilisation de la bibliothèque express-fileupload, un middleware simple conçu spécifiquement pour simplifier le processus de téléchargement de fichiers dans des applications Express.js.

Pour l'utiliser, commencez par l'installer en exécutant la commande suivante :

```
npm i express-fileupload
```

Configuration du middleware

Une fois la bibliothèque installée, dans le fichier d'entrée de l'application, il est nécessaire de la configurer afin que les fichiers téléchargés soient accessibles dans la requête via l'objet `req.files`.

```
// server.js
// 1. importer express-fileupload
const fileUpload = require("express-fileupload");

// 2. Configuration de la bibliothèque sur l'application
app.use(fileUpload());
```

Assurez-vous d'importer ce middleware après les autres middlewares tels que `express.json()` et `express.urlencoded({ extended: true })`. En effet, les middlewares dans Express s'exécutent de manière séquentielle, l'un après l'autre. Vous pouvez consulter l'article sur le chaînage des middlewares pour plus de détails.

Upload classique

Pour tester le téléchargement de fichiers, nous allons créer une route permettant d'enregistrer un fichier sur le disque dur.

Dans le répertoire des routes, nous allons créer un nouveau fichier nommé `upload.routes.js`. À l'intérieur de ce fichier, une seule route en méthode POST sera définie pour l'envoi d'un fichier:

```
const express = require("express")
const upload_routes = express.Router("")
const upload_controller = require("../controllers/upload.controller")

upload_routes.post("/fichier", upload_controller.uploadFicher)

module.exports = upload_routes
```

Dans le fichier d'entrée de l'application, nous allons enregistrer un nouveau middleware pour les routes `/upload` afin d'accéder à la route créée dans le fichier `upload.routes.js`.

```
const express = require('express');
const utilisateurs_routes = require('./routes/utilisateurs.routes');
const app = express();
const dotenv = require('dotenv');
// 1. importation du fichier contenant les routes uploads
const upload_routes = require('./routes/upload.routes');
app.use(fileUpload());
dotenv.config()

const port = process.env.PORT;

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

```
app.use(express.urlencoded({ extended: true }));
app.use(fileUpload());

// Middleware spécifique à une route
app.use('/', utilisateurs_routes)
// 2. enregistrement du nouveau middleware pour les routes upload
app.use('/upload', upload_routes)

app.listen(port, () => {
  console.log(`Serveur écoutant sur le port ${port}`);
});
```

Dans le fichier `upload.controller.js`, voici le code complet permettant d'uploader le fichier dans le dossier public.

```
const path = require("path")

const uploadFicher = async (req, res) => {
     try {
          if (req.files) {
               const fichier = req.files.fichier
               const dossier = path.resolve('./') + path.sep + 'public' + path.sep
               const nomFichier = fichier.name
               const destination = dossier + nomFichier
               await fichier.mv(destination)
               res.status(200).json({
                    message: "Votre fichier a été enregistré"
               })
          } else {
               res.status(422).send("Aucun fichier envoyé")
          }
     } catch (error) {
          console.log(error)
          res.status(500).send("Erreur interne du serveur")
     }
}
module.exports = {
     uploadFicher
}
```

```
const fichier = req.files.fichier
```

Ici, nous avons d'abord récupéré le fichier en utilisant la clé `fichier` (vous pouvez la nommer selon vos préférences).

```
const dossier = path.resolve('./') + path.sep + 'public' + path.sep
```

Ensuite, sur cette ligne, nous avons spécifié le dossier de destination (dans notre cas, public). `path.resolve('./')` permet de préciser que nous débutons à la racine, et `path.sep` est un séparateur de dossier qui s'ajuste en fonction du système utilisé (par exemple, pour Windows et / pour Linux). > *Assurez-vous que le dossier public est déjà créé à la racine de votre application. Si ce n'est pas le cas, vous devez le créer.*

Le module path est déjà installé avec Node.js. Il n'est pas nécessaire de l'installer.

```
const nomFichier = fichier.name
const destination = dossier + nomFichier
```

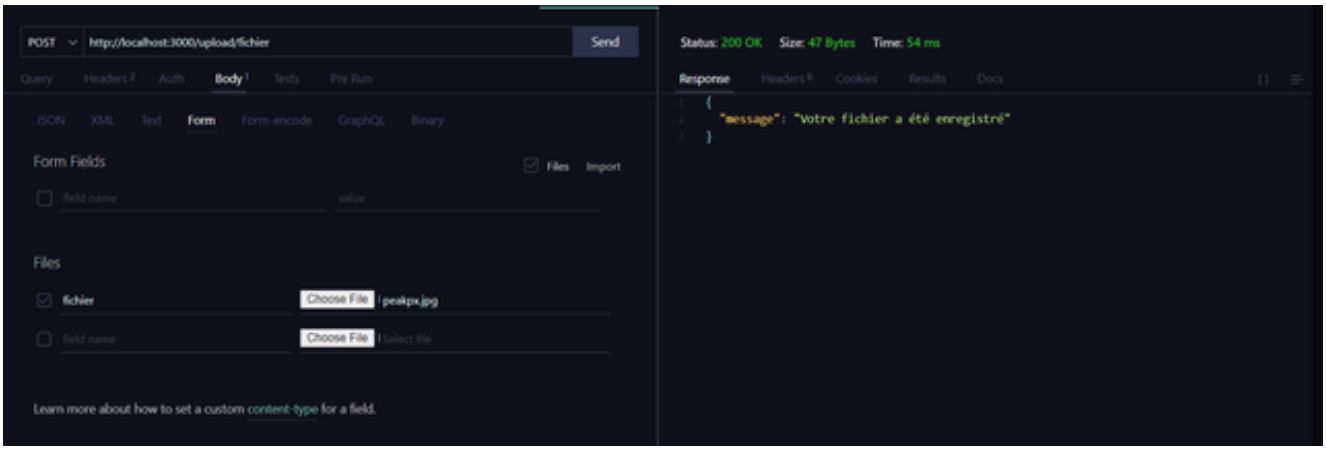
Pour ces deux lignes, nous avons récupéré le nom du fichier et concaténé le dossier de destination avec le nom du fichier afin d'obtenir le chemin final de destination du fichier.

```
await fichier.mv(destination)
```

À la fin, nous avons déplacé notre fichier vers la destination précisée. `mv` (abréviation de `move`) e est une fonction accessible sur le fichier téléchargé via express-fileupload.

Tester l'envoi des fichiers

Pour tester l'envoi des fichiers, nous allons également utiliser l'extension Thunder Client, comme illustré dans cette capture d'écran :



1. Dans le corps de la requête, sélectionnez l'option indiquant que vous envoyez des données sous forme de formulaire (Form).
2. Dans les champs de formulaire (Form Fields), cochez la case Files pour préciser que vous souhaitez envoyer des fichiers.
3. Une fois cette case cochée, vous pourrez spécifier la clé du fichier et sélectionner le fichier à envoyer.

Uploader avec une classe

Jusqu'à présent, tout fonctionne correctement, mais pour des fonctionnalités un peu plus complexes, telles que la compression des fichiers, cela pourrait devenir un peu difficile. Pour résoudre ce problème, nous avons mis à disposition une classe Upload.js qui regroupe toutes les fonctionnalités nécessaires en ce qui concerne l'upload des fichiers.

Cette classe utilise la bibliothèque sharp pour la manipulation des images. Commencez d'abord par installer la bibliothèque avec la commande suivante :

```
npm i sharp
```

Cette classe nécessite également une constante appelée `IMAGES_MIMES` que vous devez configurer dans le fichier `FILE_CONFIG.js`, situé dans le dossier `constants`. Ce fichier contient les types de fichiers que cette classe considérera comme des images. Créez le fichier et insérez-y le code suivant :

```
// constants/FILE_CONFIG.js
const IMAGES_MIMES = ['image/jpeg', 'image/gif', 'image/png']

module.exports = {
          IMAGES_MIMES
}
```

Maintenant, créez la classe Upload.js dans le dossier class et placez le code suivant :

```
const path = require('path')
const fs = require('fs')
const sharp = require('sharp')
const { IMAGES_MIMES } = require('../constants/FILE_CONFIG')

class Upload {
     constructor() {
          this.destinationPath = path.resolve('./') + path.sep + 'public' + path.sep + 'u
ploads'
     }

     async upload(file, withThumb = true, fileDestination, enableCompressing = true) {
          try {
               const extname = fileDestination ? path.extname(fileDestination) : path.ext
name(file.name)
               const defaultFileName = Date.now() + extname
               const finalFileName = fileDestination ? path.basename(fileDestination) : d
efaultFileName
               const thumbName = path.parse(finalFileName).name + '_thumb' + path.extname
(finalFileName)
               const destinationFolder = fileDestination ? path.dirname(fileDestination)
: this.destinationPath
               const filePath = destinationFolder + path.sep + finalFileName
               const thumbPath = destinationFolder + path.sep + thumbName
               if (!fs.existsSync(destinationFolder)) {
                    fs.mkdirSync(destinationFolder, { recursive: true })
               }
               const isImage = IMAGES_MIMES.includes(file.mimetype)
               var thumbInfo = undefined
               if (withThumb && isImage) {
                    thumbInfo = await sharp(file.data).resize(354, 221, { fit: 'inside' }
).toFormat('jpg').toFile(thumbPath)
               }
               var fileInfo = {}
               if (isImage && enableCompressing) {
                    fileInfo = await sharp(file.data).resize(500).toFormat(extname.substr
ing(1), { quality: 100 }).toFile(filePath.toLowerCase())
               } else {
                    fileInfo = await file.mv(filePath.toLowerCase())
               }
```

```
                return {
                     fileInfo: { ...fileInfo, fileName: finalFileName },
                     thumbInfo: withThumb ? { ...thumbInfo, thumbName } : undefined
                }
           } catch (error) {
                throw error
           }
     }
}

module.exports = Upload
```

Cette classe comporte une seule fonction `upload()` qui permet de déplacer le fichier vers le dossier précisé dans le paramètre `destinationPath` du constructeur.

Pour utiliser cette classe, vous devez créer une autre classe qui hérite de Upload afin de pouvoir changer le paramètre `destinationPath`.

Revenons à notre premier exemple de création des utilisateurs, mais cette fois nous allons ajouter la possibilité d'enregistrer une photo de l'utilisateur.

La première étape consiste à créer une constante `IMAGES_DESTINATIONS` à l'intérieur du dossier `constants`, qui nous permettra de configurer les chemins vers les téléchargements.

le fichier `IMAGES_DESTINATIONS.js` contient le code suivant:

```
// constants/IMAGES_DESTINATIONS.js
const path = require('path')

const IMAGES_DESTINATIONS = {
     utilisateurs: path.sep + 'uploads' + path.sep + 'images' + path.sep + 'utilisateurs'
}

module.exports = IMAGES_DESTINATIONS
```

Ensuite, nous allons créer une autre classe UtilisateurUpload.js dans le sous-dossier uploads situé à l'intérieur du dossier class, qui hérite de Upload.js. Ensuite, nous allons modifier le destinationPath, comme indiqué dans le code suivant :

```
// class/uploads/UtilisateurUpload.js
const IMAGES_DESTINATIONS = require("../../constants/IMAGES_DESTINATIONS")
const Upload = require("../Upload")
const path = require('path')

class UtilisateurUpload extends Upload {
     constructor() {
          super()
          this.destinationPath = path.resolve('./') + path.sep + 'public' + IMAGES_DESTIN
ATIONS.utilisateurs
     }
}
module.exports = UtilisateurUpload
```

Comme vous pouvez le constater, cette classe hérite de la classe principale `Upload` et modifie simplement la destination pour préciser l'emplacement où les téléchargements pour les utilisateurs seront stockés.

Le chemin est précisé via la constante `IMAGES_DESTINATIONS` réservée à la gestion des destinations des images.

Maintenant, nous allons ajouter une nouvelle colonne `IMAGE` à la table utilisateurs qui nous servira à enregistrer l'URL de l'image, comme illustré dans le code du modèle `Utilisateurs` :

```
// models/Utilisateurs.js

const { DataTypes } = require('sequelize')
const sequelize = require('../utils/sequelize')
const Profils = require('./Profils')

const Utilisateurs = sequelize.define('utilisateurs', {
     ID_UTILISATEUR: {
          type: DataTypes.INTEGER,
          allowNull: false,
          primaryKey: true,
          autoIncrement: true
     },
     NOM: {
          type: DataTypes.STRING(50),
          allowNull: false
     },
     PRENOM: {
          type: DataTypes.STRING(50),
          allowNull: false
     },
     ID_PROFIL: {
          type: DataTypes.INTEGER,
          allowNull: false
     },
     IMAGE: {
          type: DataTypes.STRING,
          allowNull: true,
          defaultValue: null
     },
}, {
     freezeTableName: true,
     tableName: 'utilisateurs',
     timestamps: false
})

Utilisateurs.belongsTo(Profils, { as: 'profil', foreignKey: "ID_PROFIL" })

module.exports = Utilisateurs
```

Maintenant, dans la fonction de création d'un nouvel utilisateur, nous pourrons valider et enregistrer l'image de l'utilisateur de la manière suivante :

```
// controllers/utilisateurs.controller.js
...
const UtilisateurUpload = require("../class/uploads/UtilisateurUpload")
const IMAGES_DESTINATIONS = require("../constants/IMAGES_DESTINATIONS")
const path = require("path")

const creerUtilisateur = async (req, res) => {
     try {
          const { NOM, PRENOM, ID_PROFIL } = req.body
          const { IMAGE } = req.files || {}
          const data = {
               ...req.body,
               ...req.files
          }
          const validation = new Validation(data, {
               NOM: {
                    required: true,
                    alpha: true,
                    length: [2, 20]
               },
               PRENOM: {
                    required: true,
                    alpha: true,
                    length: [2, 20]
               },
               ID_PROFIL: {
                    required: true,
                    number: true,
                    exists: "profils,ID_PROFIL"
               },
               IMAGE: {
                    required: true,
                    image: 2000000
               }
          }, {
               NOM: {
                    required: "Ce champ est obligatoire",
                    alpha: "Le nom doit contenir des caractères alphanumériques",
                    length: "Le nom doit comporter entre 2 et 20 caractères"
               },
               PRENOM: {
                    required: "Ce champ est obligatoire",
                    alpha: "Le prénom doit contenir des caractères alphanumériques",
                    length: "Le prénom doit comporter entre 2 et 20 caractères"
               },
               ID_PROFIL: {
                    required: "Le profil est obligatoire",
                    number: "Ce champ doit contenir un nombre valide",
                    exists: "Le profil n'existe pas"
               },
               IMAGE: {
                    required: "L'image de l'utilisateur est obligatoire",
                    image: "L'image est valide",
```

```
                    size: "Image trop volumineuse (max: 2Mo)"
               }
          })
          await validation.run()
          const isValid = await validation.isValidate()
          if(!isValid) {
               const errors = await validation.getErrors()
               return res.status(422).json({
                    message: "La validation des données a echouée",
                    data: errors
               })
          }
          const utilisateurUpload = new UtilisateurUpload()
          const fichier = await utilisateurUpload.upload(IMAGE)
          const imageUrl = `${req.protocol}://${req.get("host")}${IMAGES_DESTINATIONS.uti
lisateurs}${path.sep}${fichier.fileInfo.fileName}`
          const nouveauUtilisateur = await Utilisateurs.create({
               NOM: NOM,
               PRENOM: PRENOM,
               ID_PROFIL: ID_PROFIL,
               IMAGE: imageUrl
          })
          res.status(200).json({
               message: "Nouvel utilisateur créé avec succès",
               data: nouveauUtilisateur
          })
     } catch (error) {
          console.log(error)
          res.status(500).send("Erreur interne du serveur")
     }
}
...
```

Dans ce code, nous avons d'abord vérifié que nous avons reçu une image valide (ne dépassant pas 2 Mo), puis nous l'avons déplacée vers le répertoire précisé dans la classe `UtilisateurUpload`.

Dans la variable `imageUrl`, nous avons enregistré l'URL complète vers l'image pour ensuite la stocker dans la base de données. L'image enregistree sera dans le repertoire `public/uploads/images/utilisateurs`

Pour pouvoir visualiser l'image, vous devez, dans le point d'entrée de l'application, configurer en précisant le dossier où se trouvent les fichiers publics pour votre application :

```
// server.js
app.use(express.static(__dirname + "/public"));
```

Ceci permet d'indiquer que le dossier utilisé pour les fichiers publics se trouve dans le dossier `public`.

Le lien de l'image enregistrée ressemble à ceci: `http://localhost:3000\uploads\images\utilisateurs\1709880286764.jpg`. Vous pouvez la visualiser en copiant cette URL dans la barre d'adresse de votre navigateur.

Voici un tableau présentant et expliquant les paramètres que vous pouvez utiliser avec la fonction `upload()`:

Parametre	Type	Description
file	fileUpload.UploadedFile	Le fichier envoyé wt récupéré via `req.files`
withThumb	boolean	Par défaut, si vous uploadez une image, elle sera enregistrée avec sa miniature. Vous pouvez modifier ce comportement en passant `false` à ce paramètre
fileDestination	string	Par défaut, cette classe utilise le chemin défini dans `destinationPath` pour déplacer le fichier vers l'emplacement précisé. Cependant, lors de l'enregistrement direct, vous pouvez spécifier directement le chemin que vous souhaitez
enableCompressing	boolean	Cela permet de spécifier si vous souhaitez compresser les images ou non. Par défaut, si vous uploadez une image, elle sera compressée à une qualité aussi légère que possible

Authentification

Introduction

L'authentification dans une application fait référence au processus de vérification de l'identité d'un utilisateur ou d'une application qui tente d'accéder à des ressources protégées. Cela garantit que seules les parties autorisées peuvent effectuer certaines actions ou accéder à certaines données.

Il existe plusieurs méthodes d'authentification dans Express.js, mais l'une des approches courantes est l'utilisation de tokens JWT (JSON Web Tokens) et c'est ce que nous allons decouvrir dans ce chapitre.

JWT

jsonwebtokenv est un module Node.js qui permet de générer et de vérifier des JSON Web Tokens (JWT). Les JWT sont un format ouvert (RFC 7519) qui représente des informations sous la forme d'objets JSON, signés de manière cryptographique pour vérifier leur intégrité et, éventuellement, chiffrés pour assurer la confidentialité. Les JWT sont souvent utilisés pour l'authentification et l'autorisation dans les applications web et les API.

Installation du module:

```
npm install jsonwebtoken
```

Enregistrement de l'utilisateur

Lorsqu'un utilisateur s'inscrit ou s'authentifie pour la première fois, le serveur génère un JWT qui contient des informations telles que l'identifiant de l'utilisateur et son rôle. Ce token sera utilisé comme access token.

```
const jwt = require('jsonwebtoken');

const user = {
    userId: 123,
    username: 'utilisateur123',
    role: 'utilisateur',
};

const secretKey = 'votre_clé_secrète';

const accessToken = jwt.sign(user, secretKey, { expiresIn: '1h' });
```

Envoi de l'access token au client

Une fois le token généré, il est envoyé au client, généralement inclus dans la réponse à la demande d'authentification. Le client stocke ensuite ce token de manière sécurisée, par exemple dans un cookie ou dans le stockage local du navigateur.

Utilisation de l'access token dans les requêtes vers l'API

Pour accéder aux ressources protégées de l'API, le client doit inclure l'access token dans les en-têtes de ses requêtes. Ceci peut être accompli en ajoutant un en-tête d'autorisation (par exemple, Authorization: Bearer VOTRE_ACCESS_TOKEN) à chaque requête. Le serveur de l'API peut alors vérifier la validité du token à l'aide de la clé secrète partagée et autoriser ou refuser l'accès en conséquence.

Renouvellement de l'access token (si nécessaire)

Si l'access token a une durée de vie limitée et expire, le client peut demander un nouveau token en utilisant un refresh token ou en demandant à l'utilisateur de s'authentifier à nouveau.

Il est nécessaire de protéger la clé secrète utilisée pour signer les tokens et de mettre en œuvre des mécanismes de sécurité tels que HTTPS pour assurer la confidentialité des données transitant entre le client et le serveur.

Inscription de l'utilisateur

Afin de comprendre l'utilisation des JWT dans le cadre de cette formation, nous maintiendrons l'utilisation de la table `utilisateurs`. Pour ce faire, nous prévoyons d'ajouter deux nouvelles colonnes à la table à savoir la colonne `EMAIL` et la colonne `MOT_DE_PASSE`. Ces colonnes sont destinées à faciliter l'enregistrement des identifiants de l'utilisateur.

```
// // models/Utilisateurs.js
const { DataTypes } = require('sequelize');
const sequelize = require('../utils/sequelize');
const Profils = require('./Profils');
```

```
const Utilisateurs = sequelize.define('utilisateurs', {
     ID_UTILISATEUR: {
          type: DataTypes.INTEGER,
          allowNull: false,
          primaryKey: true,
          autoIncrement: true
     },
     NOM: {
          type: DataTypes.STRING(50),
          allowNull: false
     },
     PRENOM: {
          type: DataTypes.STRING(50),
          allowNull: false
     },
     ID_PROFIL: {
          type: DataTypes.INTEGER,
          allowNull: false
     },
     IMAGE: {
          type: DataTypes.STRING,
          allowNull: true,
          defaultValue: null
     },
     EMAIL: {
          type: DataTypes.STRING,
          allowNull: false,
          defaultValue: null
     },
     MOT_DE_PASSE: {
          type: DataTypes.STRING,
          allowNull: false,
          defaultValue: null
     },
}, {
     freezeTableName: true,
     tableName: 'utilisateurs',
     timestamps: false
})

Utilisateurs.belongsTo(Profils, { as: 'profil', foreignKey: "ID_PROFIL" })
```

```
module.exports = Utilisateurs
```

Dans le contrôleur, nous procéderons d'abord à la récupération et à la validation de l'email (EMAIL) et du mot de passe (MOT_DE_PASSE). Une fois que l'utilisateur est enregistré, nous générerons un jeton d'accès (access token) que nous inclurons ensuite dans la réponse. Voici à quoi ressemble la fonction de création d'un nouvel utilisateur :

```
// controllers/utilisateurs.controller.js
...
const jwt = require("jsonwebtoken")
```

```
const bcrypt = require("bcrypt")
const dotenv = require("dotenv")
dotenv.config()

const creerUtilisateur = async (req, res) => {
     try {
          const { NOM, PRENOM, ID_PROFIL, EMAIL, MOT_DE_PASSE } = req.body
          const { IMAGE } = req.files || {}
          const data = {
               ...req.body,
               ...req.files
          }
          const validation = new Validation(data, {
               NOM: {
                    required: true,
                    alpha: true,
                    length: [2, 20]
               },
               PRENOM: {
                    required: true,
                    alpha: true,
                    length: [2, 20]
               },
               ID_PROFIL: {
                    required: true,
                    number: true,
                    exists: "profils,ID_PROFIL"
               },
               IMAGE: {
                    required: true,
                    image: 2000000
               },
               EMAIL: {
                    required: true,
                    email: true,
                    unique: "utilisateurs,EMAIL"
               },
               MOT_DE_PASSE: {
                    required: true,
                    length: [8]
               }
          }, {
               NOM: {
                    required: "Ce champ est obligatoire",
                    alpha: "Le nom doit contenir des caractères alphanumériques",
                    length: "Le nom doit comporter entre 2 et 20 caractères"
               },
               PRENOM: {
                    required: "Ce champ est obligatoire",
                    alpha: "Le prénom doit contenir des caractères alphanumériques",
                    length: "Le prénom doit comporter entre 2 et 20 caractères"
               },
               ID_PROFIL: {
```

```
                    required: "Le profil est obligatoire",
                    number: "Ce champ doit contenir un nombre valide",
                    exists: "Le profil n'existe pas"
               },
               IMAGE: {
                    required: "L'image de l'utilisateur est obligatoire",
                    image: "L'image est valide",
                    size: "Image trop volumineuse (max: 2Mo)"
               },
               EMAIL: {
                    required: "L'email est obligatoire",
                    email: "Email invalide",
                    unique: "Email déjà utilisé"
               },
               MOT_DE_PASSE: {
                    required: "Le mot de passe est obligatoire",
                    length: "Le mot de passe doit contenir au moins 8 caracteres"
               }
          })
          await validation.run()
          const isValid = await validation.isValidate()
          if(!isValid) {
               const errors = await validation.getErrors()
               return res.status(422).json({
                    message: "La validation des données a echouée",
                    data: errors
               })
          }
          const utilisateurUpload = new UtilisateurUpload()
          const fichier = await utilisateurUpload.upload(IMAGE)
          const imageUrl = `${req.protocol}://${req.get("host")}${IMAGES_DESTINATIONS.uti
lisateurs}${path.sep}${fichier.fileInfo.fileName}`
          const salt = await bcrypt.genSalt()
          const password = await bcrypt.hash(MOT_DE_PASSE, salt)
          const nouveauUtilisateur = await Utilisateurs.create({
               NOM: NOM,
               PRENOM: PRENOM,
               ID_PROFIL: ID_PROFIL,
               IMAGE: imageUrl,
               EMAIL: EMAIL,
               MOT_DE_PASSE: password
          })
          const payload = {
                    ID_UTILISATEUR: nouveauUtilisateur.toJSON().ID_UTILISATEUR
          }
          const accessToken = jwt.sign(payload, process.env.JWT_PRIVATE_KEY, { expiresIn:
259200 })
          const { MOT_DE_PASSE: mdp, ...public } = utilisateur.toJSON()
          res.status(200).json({
               message: "Nouvel utilisateur créé avec succès",
               data: {
                    ...public,
                    token: accessToken
```

```
                }
            })
        } catch (error) {
            console.log(error)
            res.status(500).send("Erreur interne du serveur")
        }
}
...
```

Dans ce code, nous avons utilisé bcrypt pour crypter le mot de passe de l'utilisateur

Comme vous pouvez le remarquer, pour générer un jeton (avec la méthode `jwt.sign()`), nous avons passé un payload qui contient les données à stocker dans le jeton (`ID_UTILISATEUR` dans notre cas). Ensuite, nous avons récupéré la clé secrète pour le cryptage des jetons que nous avons placée dans le fichier `.env` avec le paramètre `JWT_PRIVATE_KEY`.

Il est strictement déconseillé de mettre des informations sensibles dans le payload, comme le mot de passe de l'utilisateur. Dans cet exemple, nous avons défini un temps d'expiration du jeton à 259200 secondes (soit 3 jours). Cela indique que le jeton ne sera plus valide après 3 jours.

Il est strictement déconseillé de mettre un délai plus long pour un token d'access

Maintenant, si vous testez la route de création d'un utilisateur, vous remarquerez que la réponse envoyée contient également le jeton d'accès.

Connexion de l'utilisateur

Le but de la connexion est de permettre aux utilisateurs déjà inscrits de se connecter. Pour ce faire, l'utilisateur doit envoyer son adresse électronique et son mot de passe afin de vérifier l'existence d'un utilisateur correspondant à ces informations. Une fois que l'utilisateur est trouvé et validé, un nouveau jeton d'accès sera généré, qu'il pourra utiliser pour accéder aux ressources sécurisées.

La route de connexion sera enregistrée dans le module auth dans le fichier d'entrée de l'application.

```
// routes/auth.routes.js
const express = require("express")
const auth_routes = express.Router("")
const auth_controller = require("../controllers/auth.controller")

auth_routes.post("/login", auth_controller.login)

module.exports = auth_routes

// server.js
const express = require('express');
const utilisateurs_routes = require('./routes/utilisateurs.routes');
const app = express();
const dotenv = require('dotenv');
const upload_routes = require('./routes/upload.routes');
const fileUpload = require("express-fileupload");
const auth_routes = require('./routes/auth.routes');
dotenv.config()

const port = process.env.PORT;
```

```
app.use(express.static(__dirname + "/public"));
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.use(fileUpload());
// app.use(monMiddleware);

// Middleware spécifique à une route
app.use('/', utilisateurs_routes)
app.use('/upload', upload_routes)
app.use('/auth', auth_routes)

app.listen(port, () => {
  console.log(`Serveur écoutant sur le port ${port}`);
});
```

```
// controllers/auth.controller.js
const Validation = require("../class/Validation")
const Utilisateurs = require("../models/Utilisateurs")
const bcrypt = require("bcrypt")
const jwt = require("jsonwebtoken")
const dotenv = require("dotenv")
dotenv.config()

const login = async (req, res) => {
     try {
          const { EMAIL, MOT_DE_PASSE } = req.body
          const validation = new Validation(req.body, {
               EMAIL: {
                    required: true,
                    email: true
               },
               MOT_DE_PASSE: {
                    required: true
               }
          }, {
               EMAIL: {
                    required: "L'email est obligatoire",
                    email: "Email invalide"
               },
               MOT_DE_PASSE: {
                    required: "Le mot de passe est obligatoire"
               }
          })
          await validation.run()
          const isValid = await validation.isValidate()
          if (!isValid) {
               const errors = await validation.getErrors()
               return res.status(422).json({
                    message: "La validation des données a echouée",
                    data: errors
               })
          }
```

```
          const utilisateur = await Utilisateurs.findOne({
               where: {
                    EMAIL: EMAIL
               }
          })
          if(utilisateur) {
               const isPasswordValid = await bcrypt.compare(MOT_DE_PASSE, utilisateur.toJ
SON().MOT_DE_PASSE)
               if(isPasswordValid) {
                    const payload = {
                              ID_UTILISATEUR: utilisateur.toJSON().ID_UTILISATEUR
                    }
                    const accessToken = jwt.sign(payload, process.env.JWT_PRIVATE_KEY, {
expiresIn: 259200 })
                    const { MOT_DE_PASSE: mdp, ...public } = utilisateur.toJSON()
                    res.status(200).json({
                         message: "Identifiants correct",
                         data: {
                              ...public,
                              token: accessToken
                         }
                    })
               } else {
                    res.status(403).json({
                         message: "Identifiants incorrects"
                    })
               }
          } else {
               res.status(403).json({
                    message: "Identifiants incorrects"
               })
          }

     } catch (error) {
          console.log(error)
          res.status(500).send("Erreur interne du serveur")
     }
}

module.exports = {
     login
}
```

À l'intérieur de ce contrôleur, une seule fonction nommée login vérifie l'authenticité de l'utilisateur. Si les identifiants fournis sont corrects, cela déclenche la génération d'un nouveau jeton d'accès.

Valider un token d'access et lier un utilisateur

Maintenant que le client a obtenu le jeton (après inscription ou connexion), ce jeton sera passé dans les en-têtes (headers) de chaque requête, et notre application procédera à la vérification de la validité du jeton. Pour simplifier cela, nous allons créer un middleware qui sera exécuté pour chaque requête envoyée à notre application.

Créez un fichier nommé bindUser.js à l'intérieur du dossier middlewares et insérez le code suivant :

```
// middlewares/bindUser.js
const jwt = require("jsonwebtoken");

const bindUser = (request, response, next) => {
     const bearer = request.headers.authorization;
     const bearerToken = bearer && bearer.split(" ")[1];
     const token = bearerToken
     if (token) {
          jwt.verify(token, process.env.JWT_PRIVATE_KEY, (error, user) => {
               if (error) {
                    next();
               } else {
                    request.userId = user.ID_UTILISATEUR;
                    next();
               }
          });
     } else {
          next();
     }
};
module.exports = bindUser;
```

Ce code vérifie que le jeton passé dans les en-têtes de la requête (avec la clé Authorization) est valide.

Si le jeton est valide, nous ajouterons `userId` à la requête, ce qui nous permettra d'indiquer que le client exécutant la requête est connecté et authentifié. Ensuite, nous passerons au middleware suivant en utilisant la fonction `next()`.

Si le jeton n'est pas valide, nous passerons simplement au middleware suivant en utilisant la fonction next(), sans pour autant ajouter userId à la requête. Une fois que le middleware est créé, il doit être enregistré dans le point d'entrée de l'application comme suit :

```
// server.js
const express = require('express');
const utilisateurs_routes = require('./routes/utilisateurs.routes');
const app = express();
const dotenv = require('dotenv');
const upload_routes = require('./routes/upload.routes');
const fileUpload = require("express-fileupload");
const auth_routes = require('./routes/auth.routes');
const bindUser = require('./middlewares/bindUser');
dotenv.config()

const port = process.env.PORT;

app.use(express.static(__dirname + "/public"));
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.use(fileUpload());
// app.use(monMiddleware);

// Middleware spécifique à une route
```

```
app.all("*", bindUser); // middleware pour verifier le token d'access
app.use('/', utilisateurs_routes)
app.use('/upload', upload_routes)
app.use('/auth', auth_routes)

app.listen(port, () => {
  console.log(`Serveur écoutant sur le port ${port}`);
});
```

Sécuriser une ressource

Maintenant que nous sommes capables de déterminer si un utilisateur est connecté ou non lors de l'exécution d'une requête, il est temps de passer à la sécurisation des ressources sensibles. Pour simplifier ce processus, nous allons créer un autre middleware appelé `requireAuth` que nous utiliserons pour indiquer que la ressource nécessite une authentification pour y accéder.

Créez un fichier requireAuth.js à l'intérieur du dossier middlewares et insérez le code suivant :

```
// middlewares/requireAuth.js
const requireAuth = (request, response, next) => {
  if (request.userId) {
    next();
  } else {
    response.status(401).json({
      errors: {
        main: "Jeton d'authentification manquante ou invalide",
      },
      authStatus: request.authStatus
    });
  }
};

module.exports = requireAuth;
```

Ce code vérifie simplement que la clé userId, que nous avons ajoutée via `bindUser`, existe dans la requête. Si elle existe, nous passerons au middleware suivant en utilisant la fonction `next()`, sinon nous bloquerons la suite en renvoyant une réponse avec le statut 401.

Maintenant que le middleware est créé, nous pourrons l'utiliser chaque fois que nous voulons sécuriser une ressource. Prenons, par exemple, le cas où nous voulons exiger une connexion pour accéder à la liste des utilisateurs. Nous pourrions le faire de la manière suivante :

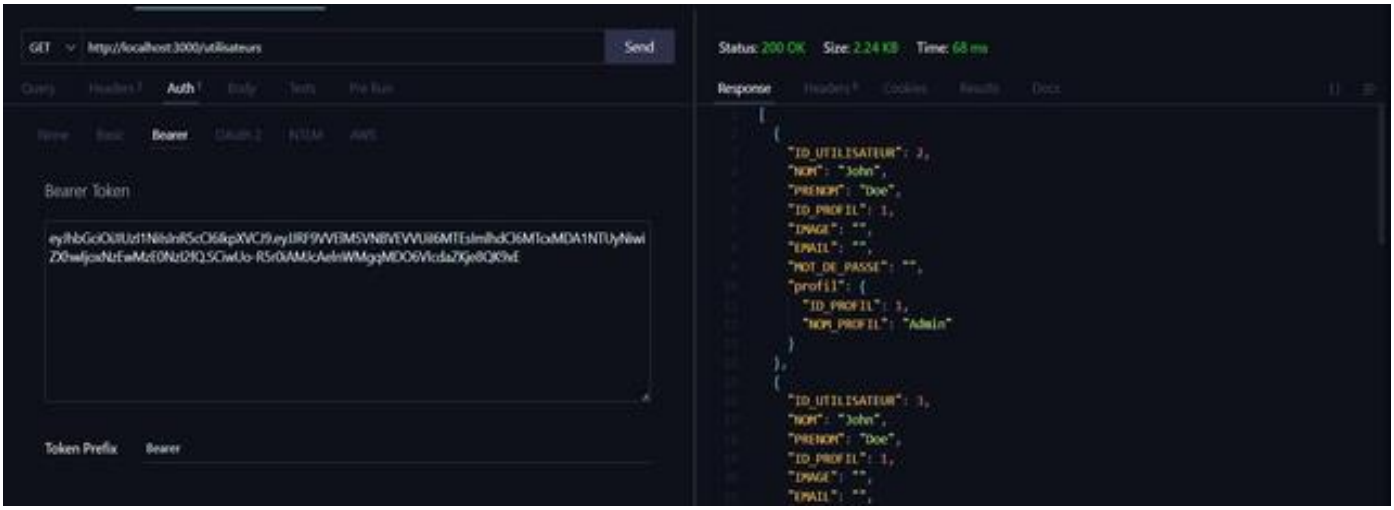
```
const express = require("express")
const utilisateurs_routes = express.Router("")
const utilisateurs_controller = require("../controllers/utilisateurs.controller")
const requireAuth = require("../middlewares/requireAuth")

utilisateurs_routes.get("/utilisateurs", requireAuth, utilisateurs_controller.getUtilisat
eurs)
utilisateurs_routes.post("/utilisateurs", utilisateurs_controller.creerUtilisateur)
utilisateurs_routes.get("/utilisateurs/:ID_UTILISATEUR", requireAuth, utilisateurs_contro
ller.findByid)
utilisateurs_routes.put("/utilisateurs/:ID_UTILISATEUR", requireAuth, utilisateurs_contro
ller.modifierUtilisateur)
```

```
utilisateurs_routes.delete("/utilisateurs/:ID_UTILISATEUR", requireAuth, utilisateurs_con
troller.supprimerUtilisateur)

module.exports = utilisateurs_routes
```

Si vous accédez maintenant à la route de récupération des utilisateurs, vous recevrez une réponse indiquant que vous devez vous connecter.

Envoyer un token d'access avec Thunder Client

Pour envoyer un jeton d'accès avec le client Thunder, commencez d'abord par vous connecter. Une fois que vous avez obtenu le jeton d'accès, vous le transmettez en tant qu'en-tête de la requête, comme illustré dans cette image :



Travail pratique

Maintenant que nous avons couvert tous les éléments nécessaires pour créer une API REST avec Express.js et Node.js, cette section est dédiée à la mise en pratique de tout ce que nous avons vu dans cette formation à travers un travail pratique.

Le travail consiste en la création d'une petite application de publication de contenu.

Voici à quoi doit ressembler l'application :

1. L'utilisateur doit d'abord créer un compte qui lui donnera accès à la publication de contenu. Chaque utilisateur doit renseigner ses informations d'identification, y compris la date de naissance et une image.
2. Une fois le compte créé, il aura également la possibilité de se connecter à son compte via son adresse e-mail et son mot de passe.
3. Une fois connecté, il pourra visualiser les publications qu'il a publiées.
4. L'utilisateur aura également la possibilité de créer, modifier ou supprimer ses propres publications. Chaque publication aura un titre, une description, une date de publication et une image.

5. Les routes de récupération et de gestion des publications doivent être protégées, et l'utilisateur aura uniquement la possibilité de gérer ses propres publications. Il n'aura pas le droit de voir les publications des autres utilisateurs.

Conclusion

Dans cette formation, nous avons appris à utiliser Node.js et Express.js pour créer une application de type API. Nous avons abordé les bases de Node.js, les avantages de l'utilisation du framework Express, ainsi que les bibliothèques utilitaires importantes qui facilitent la création d'une application Express.

Nous avons appris à utiliser Sequelize, qui simplifie les opérations d'accès à la base de données. En utilisant cette bibliothèque, l'accès à la base de données devient plus facile par rapport à l'exécution des requêtes classiques, tout en assurant également la sécurité de l'application.

Nous avons également vu comment ajouter l'authentification dans l'application avec JWT et sécuriser les ressources pour les utilisateurs authentifiés uniquement. Nous avons appris à utiliser bcrypt pour le hachage des mots de passe.

Cette formation avait pour objectif de vous donner des directives et des bonnes pratiques pour créer une application Node.js et Express.js. Il est conseillé de continuer à faire des recherches pour approfondir vos connaissances sur chacune des technologies mentionnées dans ce manuel.