*Passez au Full Stack avec Node.js, Express et MongoDB*

JavaScript reste-t-il uniquement un langage de front-end ? Plus maintenant ! Avec Node, Express et MongoDB, vous pouvez utiliser vos compétences de ninja JavaScript pour le back-end !

À partir de Node et d'Express, vous découvrirez comment les applications web fonctionnent sur le back-end. Vous créerez diverses routes pour les différentes fonctions à mettre à la disposition de votre API. Ensuite, vous ajouterez MongoDB, une base de données NoSQL, pour rendre votre API entièrement dynamique et vos données pleinement persistantes.

De nombreux cours de ce type vous permettent d'accéder à un framework front-end, tel que Angular, React ou Vue. Mais dans ce cours, l'application front-end est déjà créée ! De cette façon, vous pourrez vous concentrer sur la création de votre API et la regarder évoluer, en affinant vos compétences back-end et en choisissant librement votre propre parcours.

**L'application sur laquelle vous travaillerez est un magasin en ligne simple. Vous créerez une API pour Créer, Lire, Mettre à jour et Supprimer (CRUD pour l’anglais**

**Create, Read, Update and Delete) des éléments en vente.** **Vous ajouterez l'authentification à votre API pour la protéger, et vous intégrerez également la gestion des fichiers utilisateur pour autoriser les utilisateurs à télécharger des images.**

*Objectifs d'apprentissage*

À la fin de ce cours, vous serez capable de :

créer un serveur web simple avec Express ;

créer une API REST avec Node, Express et MongoDB ;

mettre en place un système d'authentification sur une application Express ;

gérer des fichiers utilisateur sur une application Express.

***Prérequis***

Une connaissance de la relation client-serveur, ainsi qu'une connaissance pratique de JavaScript, HTTP/AJAX, et Git. Si vous n'avez pas ces prérequis, nous vous recommandons de commencer par les cours suivants :

Apprenez la programmation à l'aide de JavaScript.

Écrivez JavaScript pour le Web.

Utilisez Git et GitHub pour vos projets de développement.

***Outils nécessaires :***

un éditeur de code ;

Node ;

Angular.

***Configurez votre environnement de développement***

*Installez Node*

Accédez à NodeJS.org pour télécharger puis installer la dernière version de Node. Cela a pour effet d'installer le runtime JavaScript de Node, ce qui vous permet par là même d'exécuter les serveurs Node. Cela installe également Node Package Manager ou npm , outil précieux pour l'installation des packages nécessaires à la création de vos projets.

*Clonez l'application front-end*

Il est maintenant temps de créer votre répertoire de travail pour ce cours : vous pouvez le nommer go-fullstack , par exemple.

Une fois le répertoire créé, il vous faudra cloner le code pour l'application front-end dans un sous-répertoire appelé frontend . À partir de votre répertoire de travail, entrez la commande ci-dessous : git clone <https://github.com/OpenClassrooms-Student-Center/go-fullstack-v3-fr.git> frontend

Vous pouvez ensuite procéder comme suit :

cd frontend

npm install

npm run start

Cela installera toutes les dépendances requises par l'application front-end, et lancera le serveur de développement. Désormais, si vous accédez à [http://localhost:4200](http://localhost:4200/), vous devriez voir l'interface suivante

***Démarrez votre serveur Node***

Qu'est-ce que Node ?

Avant de nous lancer dans l'écriture du code, qu'est-ce que Node ? Qu'est-ce qu'Express ? Y-a-t-il une différence ?

Node est le runtime qui permet d'écrire toutes nos tâches côté serveur, en JavaScript, telles que la logique métier, la persistance des données et la sécurité. Node ajoute également des fonctionnalités que le JavaScript du navigateur standard ne possède pas, comme par exemple l'accès au système de fichiers local.

Express est, pour faire court, un framework reposant sur Node, qui facilite la création et la gestion des serveurs Node, comme vous le verrez à mesure que nous progresserons dans ce cours.

*Initialisez votre projet*

Avant de commencer, vous pouvez initialiser un dépôt Git en exécutant git init depuis votre dossier backend . N'oubliez pas de créer ***un fichier .gitignore*** contenant la ligne ***node\_modules*** afin de ne pas envoyer ce dossier (qui deviendra volumineux) vers votre dépôt distant.

Si, contrairement à moi, vous créez le dépôt Git en premier, vous pourrez ajouter son URL distant à la configuration du projet Node pendant l'étape suivante.

*Démarrez un serveur basique*

***Pour créer un serveur Node dans votre fichier server.js , il vous faudra le code suivant :***

**const http = require('http');**

**const server = http.createServer((req, res) => {**

**res.end('Voilà la réponse du serveur !');**

**});**

**server.listen(process.env.PORT || 3000);**

Ici, vous importez le package HTTP natif de Node et l'utilisez pour créer un serveur, en passant une fonction qui sera exécutée à chaque appel effectué vers ce serveur. Cette fonction reçoit les objets request et response en tant qu'arguments. Dans cet exemple, vous utilisez la méthode end de la réponse pour renvoyer une réponse de type string à l'appelant.Node utilise le système de module CommonJS, donc pour importer le contenu d'un module JavaScript, on utilise le mot-clé require plutôt que le mot-clé import .

Ce système est particulièrement utile car il nous permet d'importer les modules de base de Node très facilement (comme le module  http ici) sans spécifier le chemin exact du fichier. Node sait qu'il doit importer un module de base quand on ne spécifie pas un chemin relatif (qui commence par  ./  ou  /  , par exemple).Vous verrez dans le prochain chapitre que nous utilisons aussi le mot-clé require pour importer nos propres fichiers en utilisant leur chemin relatif, comme "./app.js" ou même "./app"  , tout simplement – require  nous permet d'omettre l'extension  .js .

Dans la dernière ligne, vous configurez le serveur pour qu'il écoute :soit la variable d'environnement du port grâce à process.env.PORT : si la plateforme de déploiement propose un port par défaut, c'est celui-ci qu'on écoutera ;soit le port 3000, ce qui nous servira dans le cas de notre plateforme de développement.Démarrez le serveur en exécutant node server à partir de la ligne de commande. Pour vérifier qu'il envoie la réponse correcte, utilisez une fenêtre de navigateur pour accéder à [http://localhost:3000](http://localhost:3000/) (en supposant que vous ayez bien suivi les étapes ci-dessus).Vous pouvez également utiliser un outil de test tel que Postman pour effectuer une requête GET (ou tout autre type de requête d'ailleurs, car notre serveur ne fait actuellement aucune distinction !) vers la même URL : [http://localhost:3000](http://localhost:3000/) (à nouveau, en supposant que vous ayez bien suivi les étapes ci-dessus).

*Installez nodemon*

*Pour simplifier le développement Node, vous souhaiterez peut-être installer nodemon . Pour ce faire, exécutez la commande suivante :* ***npm install -g nodemon***

*Désormais, au lieu d'utiliser node server pour démarrer votre serveur, vous pouvez utiliser nodemon server . Il surveillera les modifications de vos fichiers et redémarrera le serveur lorsqu'il aura besoin d'être mis à jour. Cela vous garantit d'avoir toujours la dernière version de votre serveur dès que vous sauvegardez, sans devoir relancer manuellement le serveur ! Vous n'aurez qu'à relancer la requête vers votre serveur. Pratique, n'est-ce pas ?*

*Maintenant que vous savez démarrer un serveur de développement Node, dans le chapitre suivant, nous ajouterons Express au projet pour simplifier encore plus la création de notre API.*

***Créez une application Express***

*Installez Express*

Pour ajouter Express à votre projet, exécutez la commande suivante à partir de votre dossier backend : ***npm install express***

***Créez un fichier app.js , où vous placerez votre application Express :***

**const express = require('express');**

**const app = express();**

**module.exports = app;**

*Exécutez l'application Express sur le serveur Node*

Revenez à votre ***fichier server.js*** et modifiez-le comme suit :

**const http = require('http');**

**const app = require('./app');**

**app.set('port', process.env.PORT || 3000);**

**const server = http.createServer(app);**

**server.listen(process.env.PORT || 3000);**

Effectuer une demande vers ce serveur générera une erreur 404, car notre application n'a encore aucun moyen de répondre. Configurons une réponse simple pour nous assurer que tout fonctionne correctement, en effectuant un ajout à notre fichier app.js :

**const express = require('express');**

**const app = express();**

**app.use((req, res) => {**

**res.json({ message: 'Votre requête a bien été reçue !' });**

**});**

**module.exports = app;**

Si vous essayez d'effectuer une requête à votre serveur, vous devez récupérer un objet JSON contenant le message que nous avons spécifié.Maintenant que notre serveur Node gère correctement notre application Express, voyons comment nous pouvons ajouter des fonctionnalités à l'application.

***Ajoutez des middlewares***

***Une application Express est fondamentalement une série de fonctions appelées middleware. Chaque élément de middleware reçoit les objets request et response , peut les lire, les analyser et les manipuler, le cas échéant. Le middleware Express reçoit également la méthode next , qui permet à chaque middleware de passer l'exécution au middleware suivant. Voyons comment tout cela fonctionne.***

***La méthode app.use() vous permet d'attribuer un middleware à une route spécifique de votre application.***

Vous retrouvez ci-desous l'intégralité du code :

***const express = require('express');***

***const app = express();***

***app.use((req, res, next) => {***

***console.log('Requête reçue !');***

***next();***

***});***

***app.use((req, res, next) => {***

***res.status(201);***

***next();***

***});***

***app.use((req, res, next) => {***

***res.json({ message: 'Votre requête a bien été reçue !' });***

***next();***

***});***

***app.use((req, res, next) => {***

***console.log('Réponse envoyée avec succès !');***

***});***

***module.exports = app;***

***Cette application Express contient quatre éléments de middleware :***

***1- Le premier enregistre « Requête reçue ! » dans la console et passe l'exécution ;***

***2- Le deuxième ajoute un code d'état 201 à la réponse et passe l'exécution ;***

***3- Le troisième envoie la réponse JSON et passe l'exécution ;***

***4- Le dernier élément de middleware enregistre « Réponse envoyée avec succès ! » dans la console.***

***Il s'agit d'un serveur très simple et qui ne fait pas grand-chose pour l'instant, mais il illustre comment le middleware fonctionne dans une application Express.***

*Améliorez server.js*

Avant d'aller plus loin dans le cours, apportons quelques améliorations à notre fichier server.js, pour le rendre plus stable et approprié pour le déploiement :

***const http = require('http');***

***const app = require('./app');***

***const normalizePort = val => {***

***const port = parseInt(val, 10);***

***if (isNaN(port)) {***

***return val;***

***}***

***if (port >= 0) {***

***return port;***

***}***

***return false;***

***};***

***const port = normalizePort(process.env.PORT || '3000');***

***app.set('port', port);***

***const errorHandler = error => {***

***if (error.syscall !== 'listen') {***

***throw error;***

***}***

***const address = server.address();***

***const bind = typeof address === 'string' ? 'pipe ' + address : 'port: ' + port;***

***switch (error.code) {***

***case 'EACCES':***

***console.error(bind + ' requires elevated privileges.');***

***process.exit(1);***

***break;***

***case 'EADDRINUSE':***

***console.error(bind + ' is already in use.');***

***process.exit(1);***

***break;***

***default:***

***throw error;***

***}***

***};***

***const server = http.createServer(app);***

***server.on('error', errorHandler);***

***server.on('listening', () => {***

***const address = server.address();***

***const bind = typeof address === 'string' ? 'pipe ' + address : 'port ' + port;***

***console.log('Listening on ' + bind);***

***});***

***server.listen(port);***

Aperçu rapide de ce qui se passe ici :

la fonction normalizePort renvoie un port valide, qu'il soit fourni sous la forme d'un numéro ou d'une chaîne ;

la fonction errorHandler recherche les différentes erreurs et les gère de manière appropriée. Elle est ensuite enregistrée dans le serveur ;

un écouteur d'évènements est également enregistré, consignant le port ou le canal nommé sur lequel le serveur s'exécute dans la console.

Notre serveur de développement Node est à présent opérationnel. Vous pouvez ainsi ajouter les fonctionnalités appropriées à l'application Express.

***Créez une route GET***

Il est temps d'ajouter les fonctionnalités dont notre application front-end a besoin, et de regarder le système complet prendre forme ! Votre application front-end doit s'exécuter dans un navigateur. Exécutez donc npm run start à partir du répertoire front-end, accédez à [http://localhost:4200](http://localhost:4200/) avec votre navigateur puis cliquez sur « Parties 1 + 2 ».

*Remettre des articles en vente*

Comme vous avez pu le remarquer, l'application front-end affiche actuellement un spinner et indique une erreur dans la console. **Cela est dû au fait qu'elle tente d'accéder à notre API (qui n'existe pas encore !)** et de récupérer les articles en vente. Essayons de rendre ces articles accessibles.

Dans votre fichier app.js , remplacez tout le middleware par le suivant :

app.use('/api/stuff', (req, res, next) => {

const stuff = [

{

\_id: 'oeihfzeoi',

title: 'Mon premier objet',

description: 'Les infos de mon premier objet',

imageUrl: '<https://cdn.pixabay.com/photo/2019/06/11/18/56/camera-4267692_1280.jpg>',

price: 4900,

userId: 'qsomihvqios',

},

{

\_id: 'oeihfzeomoihi',

title: 'Mon deuxième objet',

description: 'Les infos de mon deuxième objet',

imageUrl: '<https://cdn.pixabay.com/photo/2019/06/11/18/56/camera-4267692_1280.jpg>',

price: 2900,

userId: 'qsomihvqios',

},

];

res.status(200).json(stuff);

});

La première différence que vous remarquerez est l'argument supplémentaire passé à la méthode use : nous lui passons un string, correspondant à la route pour laquelle nous souhaitons enregistrer cet élément de middleware. Dans ce cas, cette route serahttp://localhost:3000/api/stuff , car il s'agit de l'URL demandée par l'application front-end.

Dans ce middleware, nous créons un groupe d'articles avec le schéma de données spécifique requis par le front-end. Nous envoyons ensuite ces articles sous la forme de données JSON, avec un code 200 pour une demande réussie.

Si vous effectuez une demande GET vers cette route (aussi appelée endpoint) à partir de Postman, vous verrez que vous recevrez le groupe de stuff , mais que l'actualisation du navigateur ne semble pas fonctionner. Que se passe-t-il donc exactement ici ?

***Erreurs de CORS***

***CORS*** signifie ***« Cross Origin Resource Sharing »***. **Il s'agit d'un système de sécurité qui, par défaut, bloque les appels HTTP entre des serveurs différents, ce qui empêche donc les requêtes malveillantes d'accéder à des ressources sensibles.** Dans notre cas, nous avons deux origines : localhost:3000 et localhost:4200 , et nous souhaiterions qu'elles puissent communiquer entre elles. ***Pour cela, nous devons ajouter des headers à notre objet response.***

***Le CORS*** définit ***comment les serveurs et les navigateurs interagissent***, en spécifiant quelles ressources peuvent être demandées de manière légitime – par défaut, les requêtes AJAX sont interdites. Pour permettre des requêtes cross-origin (et empêcher des erreurs CORS), des headers spécifiques de contrôle d'accès doivent être précisés pour tous vos objets de réponse.

De retour au fichier app.js , ajoutez le middleware suivant avant la route d'API :

**app.use((req, res, next) => {**

**res.setHeader('Access-Control-Allow-Origin', '\*');**

**res.setHeader('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content, Accept, Content-Type, Authorization');**

**res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE, PATCH, OPTIONS');**

**next();**

**});**

**Ces headers permettent :**

**1- *d'accéder à notre API depuis n'importe quelle origine ( '\*' ) ;***

**2- *d'ajouter les headers mentionnés aux requêtes envoyées vers notre API (Origin , X-Requested-With , etc.) ;***

**3- *d'envoyer des requêtes avec les méthodes mentionnées ( GET ,POST , etc.).***

Comme vous pouvez le voir dans le code, le middleware ne prend pas d'adresse en premier paramètre, afin de s'appliquer à toutes les routes. Cela permettra à toutes les demandes de toutes les origines d'accéder à votre API. Vous pouvez également ajouter des URL d'images valides aux stuff renvoyés par l'API, en terminant la route GET. Si vous actualisez à présent l'application front-end, vous devriez voir vos articles en vente. Maintenant que vous avez correctement implémenté une route basique pour remettre les stuff en vente, voyons comment autoriser les utilisateurs à proposer leurs propres articles à la vente.

***Créez une route POST***

*Recevez des articles de l'application front-end*

Bien que nous ne puissions pas, pour le moment, stocker de données envoyées par l'utilisateur, puisque nous n'avons configuré aucune base de données, nous pouvons du moins veiller à recevoir correctement les données du front-end. L'application front-end contient un formulaire « Vendre un article », qui envoie une demande POST (contenant l'article à vendre) à notre route api/stuff . Voyons comment nous pouvons capturer ces données.

***Pour gérer la requête POST venant de l'application front-end, on a besoin d'en extraire le corps JSON.*** Pour cela, vous avez juste besoin d'un middleware très simple, mis à disposition par le framework Express. Juste après la déclaration de la constante app , ajoutez :

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

Avec ceci, Express prend toutes les requêtes qui ont comme Content-Type application/json et met à disposition leur body directement sur l'objet req, ce qui nous permet d'écrire le middleware POST suivant :

**app.post('/api/stuff', (req, res, next) => {**

**console.log(req.body);**

**res.status(201).json({**

**message: 'Objet créé !' });**

**});**

En passant votre middleware à app.post() au lieu de app.use() , il répondra uniquement aux requêtes de type POST.

**Qu'avons-nous appris dans cette première partie ?**

**Dans cette première partie du cours, vous avez :**

**configuré votre environnement de développement, avec toutes les dépendances requises pour démarrer ;**

**démarré votre premier serveur Node et l'avez utilisé pour gérer votre première application Express ;**

**créé deux routes pour votre application, et vous avez également implémenté CORS pour vous assurer que le front-end pouvait effectuer des appels vers votre application en toute sécurité.**

***Configurez votre base de données***

Mais pourquoi MongoDB ? Vous avez peut-être entendu dire qu'il s'agit d'une base de données NoSQL, mais qu'est-ce que ça veut dire ? Déjà, qu'est-ce qu'une base de données SQL ? D'ailleurs, qu'est-ce qu'une base de données tout court ?!

**Qu'est-ce qu'une base de données ?**

Une base de données, en gros, est une collection de données structurées. La structure des données, ainsi que l'organisation de la collection dépendent du type de base de données. Il y a deux types de base qui nous intéresseront ici : SQL et NoSQL.

**Qu'est-ce qu'une base de données SQL ?**

Une base de données SQL (prononcé "ci-couel", ou "sequel" en anglais) est une base de données relationnelle qui emploie le langage Structured Query Language pour gérer les données qu'elle contient. Généralement, une base de données SQL organise ses données dans des tables selon des schémas stricts. Par exemple, si on décide qu'un utilisateur a un prénom, un nom et une adresse mail, on aura une table Users avec des colonnes first\_name , last\_name et email , et chaque rangée correspondra à un utilisateur. Pour être un utilisateur valide, chaque utilisateur doit avoir un prénom, un nom et une adresse mail, donc on peut être certain que chaque utilisateur tiré de la base de données aura ces attributs.

Dans une base de données SQL, les relations entre les différentes tables sont très importantes. Pour cette raison, notre table Users aura également une colonne id pour l'identifiant individuel unique de chaque utilisateur. Cela signifie que l'on peut faire référence à un utilisateur depuis une autre table. Par exemple, on pourrait avoir une table Orders pour les commandes des utilisateurs. Chaque commande doit appartenir à un utilisateur, donc on fera référence à l'identifiant de l'utilisateur dans la colonne user\_id de la table Orders.

Les bases de données SQL (comme MySQL ou PostgreSQL) sont très bien adaptées aux données relationnelles et pour des données où des définitions fortes sont nécessaires. Cependant, dans le contexte de projets plus petits, des MVP ou des startups, on ne sait peut-être pas encore à quoi ressemblera notre dernier modèle de données. On souhaitera aussi potentiellement pouvoir grandir rapidement (il est très difficile d'agrandir une base de données SQL au-delà d'une certaine limite) pour pouvoir gérer un plus grand nombre d'utilisateurs. Voilà où le NoSQL montre son utilité, en particulier MongoDB.

**Qu'est-ce que MongoDB ?**

MongoDB est une base de données NoSQL. Cela signifie que l'on ne peut pas utiliser SQL pour communiquer avec. Les données sont stockées comme des collections de documents individuels décrits en JSON (JavaScript Object Notation). Il n'y a pas de schéma strict de données (on peut écrire, en gros, ce que l'on veut où l'on veut), et il n'y a pas de relation concrète entre les différentes données. Cependant, il existe des outils (que vous découvrirez rapidement !) pour nous aider à subvenir à ces besoins.

Les avantages principaux de MongoDB sont son évolutivité et sa flexibilité. Le site officiel décrit MongoDB comme étant "construit pour des personnes qui construisent des applications Internet et des applications métier qui ont besoin d'évoluer rapidement et de grandir élégamment". La compétence MongoDB est donc très recherchée dans les startups et PME. Un autre avantage est la facilité avec laquelle on communique avec la base de données avec JavaScript, avec les documents décrits en JSON. Cela vous permet d'appliquer les connaissances JS que vous avez déjà à la couche base de données !

**Connectez votre API à votre cluster MongoDB**

Mongoose est un package qui facilite les interactions avec notre base de données MongoDB. Il nous permet de :

**valider le format des données ;**

**gérer les relations entre les documents ;**

**communiquer directement avec la base de données pour la lecture et l'écriture des documents.**

Tout cela nous permet de dépasser plusieurs des obstacles que l'on peut rencontrer avec des bases de données NoSQL, et d'appliquer nos connaissances en JavaScript à une base encore plus fonctionnelle !

Une fois l'installation terminée, importez mongoose dans votre fichier app.js en ajoutant la constante suivante :

**const mongoose = require('mongoose');**

Juste en dessous de votre déclaration de constante app, ajoutez la ligne suivante. Veillez à remplacer l'adresse SRV par la vôtre, et la chaîne <PASSWORD> par votre mot de passe utilisateur MongoDB :

**mongoose.connect('mongodb+srv://jimbob:<PASSWORD>@cluster0-pme76.mongodb.net/test?retryWrites=true&w=majority',**

**{ useNewUrlParser: true,**

**useUnifiedTopology: true })**

**.then(() => console.log('Connexion à MongoDB réussie !'))**

**.catch(() => console.log('Connexion à MongoDB échouée !'));**

Après enregistrement, voire redémarrage de votre serveur Node si nécessaire, vous devriez voir le message « Connexion à MongoDB Atlas réussie ! » dans la console. Votre API est à présent connectée à votre base de données, et nous pouvons commencer à créer des routes serveur afin d'en bénéficier.

Pour des applications qui ont besoin d'évoluer rapidement, les bases de données NoSQL comme MongoDB sont souvent un bon choix de technologie.

MongoDB Atlas permet d'héberger gratuitement une base de données MongoDB.

Le package Mongoose facilite les interactions entre votre application Express et votre base de données MongoDB.

***Créez un schéma de données***

L'un des avantages que nous avons à utiliser Mongoose pour gérer notre base de données MongoDB est que nous pouvons implémenter des schémas de données stricts, qui permettent de rendre notre application plus robuste. Commençons par créer un schéma Thing (« chose ») pour tout objet mis en vente dans notre application.

*Créez un schéma Thing*

Dans votre dossier backend , créez un dossier appelé ***models*** et, dans ce nouveau dossier, un fichier appelé ***thing.js*** :

**const mongoose = require('mongoose');**

**const thingSchema = mongoose.Schema({**

**title: { type: String, required: true },**

**description: { type: String, required: true },**

**imageUrl: { type: String, required: true },**

**userId: { type: String, required: true },**

**price: { type: Number, required: true },**

**});**

**module.exports = mongoose.model('Thing', thingSchema);**

Ici, voici ce que nous faisons :

nous créons un schéma de données qui contient les champs souhaités pour chaque Thing, indique leur type ainsi que leur caractère (obligatoire ou non). Pour cela, on utilise la méthode Schema mise à disposition par Mongoose. Pas besoin de mettre un champ pour l'Id puisqu'il est automatiquement généré par Mongoose ;

ensuite, nous exportons ce schéma en tant que modèle Mongoose appelé « Thing », le rendant par là même disponible pour notre application Express.

Ce modèle vous permettra non seulement d'appliquer notre structure de données, mais aussi de simplifier les opérations de lecture et d'écriture dans la base de données, comme vous le verrez dans les chapitres suivants.

***La méthode Schema de Mongoose vous permet de créer un schéma de données pour votre base de données MongoDB.***

***La méthode model transforme ce modèle en un modèle utilisable.***

Maintenant que vous avez défini le schéma de données, voyons comment sauvegarder et récupérer les données de notre base de données !

***Enregistrez et récupérez des données***

En utilisant le modèle Thing que nous avons créé dans le chapitre précédent, nous allons exploiter Mongoose. Vous verrez que l'enregistrement et la récupération de données dans la base de données est un jeu d'enfant ! Commençons par implémenter correctement notre route POST.

*Enregistrement des Things dans la base de données*

Pour pouvoir utiliser notre nouveau modèle Mongoose dans l'application, nous devons l'importer dans le fichier app.js :

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

Maintenant, remplacez la logique de votre route POST par :

**app.post('/api/stuff', (req, res, next) => {**

**delete req.body.\_id;**

**const thing = new Thing({**

**...req.body**

**});**

**thing.save()**

**.then(() => res.status(201).json({ message: 'Objet enregistré !'}))**

**.catch(error => res.status(400).json({ error }));**

**});**

Ici, vous créez une instance de votre modèle Thing en lui passant un objet JavaScript contenant toutes les informations requises du corps de requête analysé (en ayant supprimé en amont le faux\_id envoyé par le front-end).

**L'opérateur spread ... est utilisé pour faire une copie de tous les éléments de req.body** . Pour plus d'informations sur l'opérateur spread, rendez-vous sur la documentation de MDN.

Ce modèle comporte une méthode save() qui enregistre simplement votre Thing dans la base de données.

La base de données MongoDB est fractionnée en collections : le nom de la collection est défini par défaut sur le pluriel du nom du modèle. Ici, ce sera Things .

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

*Récupération de la liste de Things en vente*

Désormais, nous pouvons implémenter notre route GET afin qu'elle renvoie tous les Things dans la base de données :

**app.use('/api/stuff', (req, res, next) => {**

**Thing.find()**

**.then(things => res.status(200).json(things))**

**.catch(error => res.status(400).json({ error }));**

**});**

Dans l'exemple ci-dessus, nous utilisons la méthode find() dans notre modèle Mongoose afin de renvoyer un tableau contenant tous les Things dans notre base de données. À présent, si vous ajoutez un Thing , il doit s'afficher immédiatement sur votre page d'articles en vente.

En revanche, si vous cliquez sur l'un des Things , l'affichage d'un seul élément ne fonctionne pas. En effet, il tente d'effectuer un appel GET différent pour trouver un Thing individuel. Implémentons cette route maintenant.

*Récupération d'un Thing spécifique by id*

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

**app.get('/api/stuff/:id', (req, res, next) => {**

**Thing.findOne({ \_id: req.params.id })**

**.then(thing => res.status(200).json(thing))**

**.catch(error => res.status(404).json({ error }));**

**});**

Dans cette route :

nous utilisons la méthode get() pour répondre uniquement aux demandes GET à cet endpoint ;

nous utilisons deux-points : en face du segment dynamique de la route pour la rendre accessible en tant que paramètre ;

nous utilisons ensuite la méthode **findOne()** dans notre modèle Thing pour trouver le Thing unique ayant le même \_id que le paramètre de la requête ;

ce Thing est ensuite retourné dans une Promise et envoyé au front-end ;

si aucun Thing n'est trouvé ou si une erreur se produit, nous envoyons une erreur 404 au front-end, avec l'erreur générée.

Maintenant, notre application commence vraiment à prendre forme ! Nous pouvons créer des objets et les voir apparaître immédiatement dans notre boutique en ligne grâce à la base de données. Et nous pouvons même ouvrir un objet en particulier pour obtenir les informations de cet objet précis, via la base de données.

Dans le chapitre suivant, nous implémenterons nos boutons Modifier et Supprimer pour terminer la partie Thing de notre API.

Les méthodes de votre modèle Thing permettent d'interagir avec la base de données :

**save() – enregistre un Thing ;**

**find() – retourne tous les Things ;**

**findOne() – retourne un seul Thing basé sur la fonction de comparaison qu'on lui passe (souvent pour récupérer un Thing par son identifiant unique).**

**La méthode app.get() permet de réagir uniquement aux requêtes de type GET.**

Dans le prochain chapitre, nous utiliserons le reste des opérations CRUD pour mettre à jour et supprimer nos données dans la base de données. Allons-y !

***Modifiez et supprimez des données***

*Mettez à jour un Thing existant*

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

**app.put('/api/stuff/:id', (req, res, next) => {**

**Thing.updateOne({ \_id: req.params.id }, { ...req.body, \_id: req.params.id })**

**.then(() => res.status(200).json({ message: 'Objet modifié !'}))**

**.catch(error => res.status(400).json({ error }));**

**});**

Ci-dessus, nous exploitons la méthode **updateOne()** dans notre modèle Thing . Cela nous permet de **mettre à jour le Thing qui correspond à l'objet que nous passons comme premier argument**. Nous utilisons aussi le paramètre id passé dans la demande, et le remplaçons par le Thing passé comme second argument.

**L'utilisation du mot-clé new avec un modèle Mongoose crée par défaut un champ\_id . Utiliser ce mot-clé générerait une erreur, car nous tenterions de modifier un champ immuable dans un document de la base de données. Par conséquent, nous devons utiliser le paramètre id de la requête pour configurer notre Thing avec le même \_id qu'avant.**

Vous pouvez maintenant tester votre nouvelle route : cliquez sur un Thing de l'application, puis sur son bouton Modifier, changez ses paramètres puis sauvegardez. Vous envoyez alors un Thing modifié au back-end. En revenant sur la page des articles, vous devriez retrouver votre article modifié.

*Suppression d'un Thing*

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

**app.delete('/api/stuff/:id', (req, res, next) => {**

**Thing.deleteOne({ \_id: req.params.id })**

**.then(() => res.status(200).json({ message: 'Objet supprimé !'}))**

**.catch(error => res.status(400).json({ error }));**

**});**

La méthode **deleteOne()** de notre modèle fonctionne comme **findOne()** et **updateOne()** dans le sens où nous lui passons un objet correspondant au document à supprimer. Nous envoyons ensuite une réponse de réussite (200) ou d'échec (400) au front-end.

Félicitations ! Désormais, votre application implémente **le CRUD** complet :

**create (création de ressources) ;**

**read (lecture de ressources) ;**

**update (modification de ressources) ;**

**delete (suppression de ressources).**

L'application permet donc désormais un parcours client complet !

**app.put()** et **app.delete()** attribuent des middlewares aux requêtes de **type PUT** et de **type DELETE**.

Les méthodes **updateOne()** et **delete()** de votre modèle Thing permettent de mettre à jour ou de supprimer un Thing dans la base de données.

**Qu'avons-nous appris dans cette partie du cours ?**

**Vous avez configuré votre base de données MongoDB et l'avez connectée à votre application Express.**

**Vous avez utilisé Mongoose pour créer un modèle de données afin de faciliter les opérations de la base de données.**

**Vous avez implémenté, dans votre application Express, les routes CRUD qui exploitent votre modèle de données Mongoose, rendant ainsi votre application entièrement dynamique.**

**Dans la partie suivante de ce cours, nous nous pencherons sur une thématique extrêmement importante : la sécurité.**

***2 EME PARTIE:***

***Optimisez la structure du back-end***

Avant de nous lancer dans le sujet complexe qu'est l'authentification, nous allons réorganiser la structure de notre back-end pour en faciliter la compréhension et la gestion. Il est techniquement possible de conserver toute notre logique dans le fichier app.js , mais ce fichier peut vite devenir bien trop volumineux, ce qui compliquerait la gestion. Par conséquent, rendons les choses un peu plus modulaires.

*Configurez le routage*

La première chose que nous allons faire est de dissocier notre logique de routing et la logique globale de l'application. Créez, dans votre dossier backend , un dossier routes puis, dans ce nouveau dossier, un fichier stuff.js . Celui-ci contiendra la logique de nos routes stuff :

**const express = require('express');**

**const router = express.Router();**

**module.exports = router;**

Ci-dessus, nous créons un routeur Express. Jusqu'à présent, nous avions enregistré nos routes directement dans notre application. Maintenant, nous allons les enregistrer dans notre routeur Express, puis enregistrer celui-ci dans l'application.

Il est temps de couper toutes nos routes de app.js et de les coller dans notre routeur. Veillez à remplacer toutes les occurrences de app par router , car nous enregistrons les routes dans notre routeur :

**const express = require('express');**

**const router = express.Router();**

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

**router.post('/', (req, res, next) => {**

**const thing = new Thing({**

**title: req.body.title,**

**description: req.body.description,**

**imageUrl: req.body.imageUrl,**

**price: req.body.price,**

**userId: req.body.userId**

**});**

**thing.save().then(**

**() => {**

**res.status(201).json({**

**message: 'Post saved successfully!'**

**});**

**}**

**).catch(**

**(error) => {**

**res.status(400).json({**

**error: error**

**});**

**}**

**);**

**});**

**router.get('/:id', (req, res, next) => {**

**Thing.findOne({**

**\_id: req.params.id**

**}).then(**

**(thing) => {**

**res.status(200).json(thing);**

**}**

**).catch(**

**(error) => {**

**res.status(404).json({**

**error: error**

**});**

**}**

**);**

**});**

**router.put('/:id', (req, res, next) => {**

**const thing = new Thing({**

**\_id: req.params.id,**

**title: req.body.title,**

**description: req.body.description,**

**imageUrl: req.body.imageUrl,**

**price: req.body.price,**

**userId: req.body.userId**

**});**

**Thing.updateOne({\_id: req.params.id}, thing).then(**

**() => {**

**res.status(201).json({**

**message: 'Thing updated successfully!'**

**});**

**}**

**).catch(**

**(error) => {**

**res.status(400).json({**

**error: error**

**});**

**}**

**);**

**});**

**router.delete('/:id', (req, res, next) => {**

**Thing.deleteOne({\_id: req.params.id}).then(**

**() => {**

**res.status(200).json({**

**message: 'Deleted!'**

**});**

**}**

**).catch(**

**(error) => {**

**res.status(400).json({**

**error: error**

**});**

**}**

**);**

**});**

**router.get('/' +**

**'', (req, res, next) => {**

**Thing.find().then(**

**(things) => {**

**res.status(200).json(things);**

**}**

**).catch(**

**(error) => {**

**res.status(400).json({**

**error: error**

**});**

**}**

**);**

**});**

**module.exports = router;**

/api/stuff doit être supprimé de chaque segment de route. Si cela supprime une chaîne de route, veillez à laisser une barre oblique / (reportez-vous au code ci-dessus).

Veillez également à remplacer l'élément app.use() final par app.get() si ce n'est pas déjà fait, car cette route ne concerne que les demandes GET.

Nous devons désormais enregistrer notre nouveau routeur dans notre fichier app.js . D'abord, nous devons l'importer :

**const stuffRoutes = require('./routes/stuff');**

Nous l'enregistrerons ensuite comme nous le ferions pour une route unique. Nous voulons enregistrer notre routeur pour toutes les demandes effectuées vers /api/stuff . Par conséquent, tapez :

**app.use('/api/stuff', stuffRoutes);**

Restez dans la section « Parties 1 + 2 » pour le moment. Si vous actualisez l'application front-end, tout devrait encore fonctionner comme avant.

*Configurez les contrôleurs*

Pour rendre notre structure encore plus modulaire, et simplifier la lecture et la gestion de notre code, nous allons séparer la logique métier de nos routes en contrôleurs.

Créez un dossier controllers dans votre dossier backend et créez un autre fichier stuff.js . Celui-ci sera notre contrôleur stuff . Copions le premier élément de logique métier de la route POST vers notre contrôleur :

**// in controllers/stuff.js**

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

**exports.createThing = (req, res, next) => {**

**const thing = new Thing({**

**title: req.body.title,**

**description: req.body.description,**

**imageUrl: req.body.imageUrl,**

**price: req.body.price,**

**userId: req.body.userId**

**});**

**thing.save().then(**

**() => {**

**res.status(201).json({**

**message: 'Post saved successfully!'**

**});**

**}**

**).catch(**

**(error) => {**

**res.status(400).json({**

**error: error**

**});**

**}**

**);**

**};**

**Ici, nous exposons la logique de notre route POST en tant que fonction appelée createThing() . Pour réimplémenter cela dans notre route, nous devons importer notre contrôleur puis enregistrer createThing :**

**// in routes/stuff.js**

**const stuffCtrl = require('../controllers/stuff');**

**router.get('/', stuffCtrl.getAllStuff);**

**Nous pouvons maintenant faire de même pour toutes nos autres routes. Voici le contrôleur final :**

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

**exports.createThing = (req, res, next) => {**

**const thing = new Thing({**

**title: req.body.title,**

**description: req.body.description,**

**imageUrl: req.body.imageUrl,**

**price: req.body.price,**

**userId: req.body.userId**

**});**

**thing.save().then(**

**() => {**

**res.status(201).json({**

**message: 'Post saved successfully!'**

**});**

**}**

**).catch(**

**(error) => {**

**res.status(400).json({**

**error: error**

**});**

**}**

**);**

**};**

**exports.getOneThing = (req, res, next) => {**

**Thing.findOne({**

**\_id: req.params.id**

**}).then(**

**(thing) => {**

**res.status(200).json(thing);**

**}**

**).catch(**

**(error) => {**

**res.status(404).json({**

**error: error**

**});**

**}**

**);**

**};**

**exports.modifyThing = (req, res, next) => {**

**const thing = new Thing({**

**\_id: req.params.id,**

**title: req.body.title,**

**description: req.body.description,**

**imageUrl: req.body.imageUrl,**

**price: req.body.price,**

**userId: req.body.userId**

**});**

**Thing.updateOne({\_id: req.params.id}, thing).then(**

**() => {**

**res.status(201).json({**

**message: 'Thing updated successfully!'**

**});**

**}**

**).catch(**

**(error) => {**

**res.status(400).json({**

**error: error**

**});**

**}**

**);**

**};**

**exports.deleteThing = (req, res, next) => {**

**Thing.deleteOne({\_id: req.params.id}).then(**

**() => {**

**res.status(200).json({**

**message: 'Deleted!'**

**});**

**}**

**).catch(**

**(error) => {**

**res.status(400).json({**

**error: error**

**});**

**}**

**);**

**};**

**exports.getAllStuff = (req, res, next) => {**

**Thing.find().then(**

**(things) => {**

**res.status(200).json(things);**

**}**

**).catch(**

**(error) => {**

**res.status(400).json({**

**error: error**

**});**

**}**

**);**

**};**

Voici notre routeur final :

**const express = require('express');**

**const router = express.Router();**

**const stuffCtrl = require('../controllers/stuff');**

**router.get('/', stuffCtrl.getAllStuff);**

**router.post('/', stuffCtrl.createThing);**

**router.get('/:id', stuffCtrl.getOneThing);**

**router.put('/:id', stuffCtrl.modifyThing);**

**router.delete('/:id', stuffCtrl.deleteThing);**

**module.exports = router;**

Comme vous le voyez, cela facilite la compréhension de notre fichier de routeur. Quelles routes sont disponibles à quels points de terminaison est évident, et les noms descriptifs donnés aux fonctions de notre contrôleur permettent de mieux comprendre la fonction de chaque route.

Structurer le code de manière modulaire comme cela n'est pas absolument nécessaire pour chaque projet, mais c'est une bonne habitude à prendre car cela simplifie la maintenance.

La méthode ***express.Router()*** vous permet de créer des routeurs séparés pour chaque route principale de votre application – vous y enregistrez ensuite les routes individuelles.

Un fichier de contrôleur exporte des méthodes qui sont ensuite attribuées aux routes pour améliorer la maintenabilité de votre application.

Maintenant que tout est prêt, commençons à implémenter l'authentification utilisateur.

***Préparez la base de données pour les informations d'authentification***

*Comprenez le stockage de mot de passe sécurisé*

Dans les chapitres suivants, nous implémenterons l'authentification par e-mail et mot de passe pour notre API. Cela implique de stocker des mots de passe utilisateur dans notre base de données d'une manière ou d'une autre. Ce que nous ne voulons certainement pas faire est de les stocker sous la forme de texte brut : quiconque accéderait à notre base de données verrait la liste complète des informations de connexion de tous les utilisateurs. À la place, nous stockerons le mot de passe de chaque utilisateur sous la forme d'un hash ou d'une chaîne chiffrée.

Le package de chiffrement que nous utiliserons, bcrypt , utilise un algorithme unidirectionnel pour chiffrer et créer un hash des mots de passe utilisateur, que nous stockerons ensuite dans le document de la base de données relatif à chaque utilisateur. Lorsqu'un utilisateur tentera de se connecter, nous utiliserons bcrypt pour créer un hash avec le mot de passe entré, puis le comparerons au hash stocké dans la base de données. Ces deux hashs ne seront pas les mêmes : cela poserait un problème de sécurisation, car les pirates informatiques n'auraient qu'à deviner les mots de passe jusqu'à ce que les hashs correspondent. Le package bcrypt permet d'indiquer si les deux hashs ont été générés à l'aide d'un même mot de passe initial. Il nous aidera donc à implémenter correctement le stockage et la vérification sécurisés des mots de passe.

La première étape de l'implémentation de l'authentification est de créer un modèle de base de données pour les informations de nos utilisateurs.

*Créez un modèle de données*

Pour s'assurer que deux utilisateurs ne puissent pas utiliser la même adresse e-mail, nous utiliserons le mot clé unique pour l'attribut email du schéma d'utilisateur userSchema . Les erreurs générées par défaut par MongoDB pouvant être difficiles à résoudre, nous installerons un package de validation pour prévalider les informations avant de les enregistrer :

***npm install mongoose-unique-validator***

Il se peut que vous ayez besoin d'ajouter le flag --force à la commande ci-dessus si jamais vous installez le validateur peu de temps après l'arrivée d'une nouvelle version de mongoose : parfois les personnes qui s'occupent du package validateur mettent un peu de temps pour le mettre à jour. Mais ne nous frustrons pas : ils font un super travail d'open source bénévole !

Ce package une fois installé, nous pouvons créer notre propre modèle utilisateur :

**const mongoose = require('mongoose');**

**const uniqueValidator = require('mongoose-unique-validator');**

**const userSchema = mongoose.Schema({**

**email: { type: String, required: true, unique: true },**

**password: { type: String, required: true }**

**});**

**userSchema.plugin(uniqueValidator);**

**module.exports = mongoose.model('User', userSchema);**

Dans notre schéma, la valeur unique , avec l'élément ***mongoose-unique-validator*** passé comme plug-in, s'assurera que deux utilisateurs ne puissent partager la même adresse e-mail.

***bcrypt est un package de cryptage que vous pouvez installer avec npm .***

***mongoose-unique-validator*** améliore les messages d'erreur lors de l'enregistrement de données uniques.

Maintenant que notre modèle est prêt, nous allons commencer à l'utiliser dans le chapitre suivant pour enregistrer les nouveaux utilisateurs dans notre base de données, et appliquer le chiffrement de mot de passe.

***Créez des utilisateurs***

*Configurez les routes d'authentification*

Commençons par créer l'infrastructure nécessaire à nos routes d'authentification. Il nous faudra un contrôleur et un routeur, puis nous devrons enregistrer ce routeur dans notre application Express.

D'abord, créez un fichier user.js dans votre dossier controllers :

**exports.signup = (req, res, next) => {**

**};**

**exports.login = (req, res, next) => {**

**};**

Nous implémenterons ces fonctions bientôt. Pour l'instant, terminons la création des routes.

Pour ce faire, créez un autre fichier user.js , cette fois dans votre dossier routes :

**const express = require('express');**

**const router = express.Router();**

**const userCtrl = require('../controllers/user');**

**router.post('/signup', userCtrl.signup);**

**router.post('/login', userCtrl.login);**

**module.exports = router;**

Les routes fournies sont celles prévues par l'application front-end.

**N'oubliez pas que le segment de route indiqué ici est uniquement le segment final, car le reste de l'adresse de la route sera déclaré dans notre application Express.**

Maintenant, enregistrons notre routeur dans notre application. Pour ce faire, importez le routeur :

**const userRoutes = require('./routes/user');**

Puis enregistrez-le :

**app.use('/api/stuff', stuffRoutes);**

**app.use('/api/auth', userRoutes);**

Nos routes sont maintenant prêtes. Il est temps de commencer l'implémentation de la logique métier.

*Créez des utilisateurs*

Il nous faudra le package de chiffrement bcrypt pour notre fonction signup . Installons-le donc dans notre projet :

**npm install bcrypt**

Nous pouvons l'importer dans notre contrôleur et implémenter notre fonction signup (n'oubliez pas d'importer votre modèle User !) :

**exports.signup = (req, res, next) => {**

**bcrypt.hash(req.body.password, 10)**

**.then(hash => {**

**const user = new User({**

**email: req.body.email,**

**password: hash**

**});**

**user.save()**

**.then(() => res.status(201).json({ message: 'Utilisateur créé !' }))**

**.catch(error => res.status(400).json({ error }));**

**})**

**.catch(error => res.status(500).json({ error }));**

**};**

Dans cette fonction :

nous appelons la fonction de hachage de bcrypt dans notre mot de passe et lui demandons de « saler » le mot de passe 10 fois. Plus la valeur est élevée, plus l'exécution de la fonction sera longue, et plus le hachage sera sécurisé. Pour plus d'informations, consultez la documentation de bcrypt ;

il s'agit d'une fonction asynchrone qui renvoie une Promise dans laquelle nous recevons le hash généré ;

dans notre bloc then , nous créons un utilisateur et l'enregistrons dans la base de données, en renvoyant une réponse de réussite en cas de succès, et des erreurs avec le code d'erreur en cas d'échec.

***La méthode hash() de bcrypt crée un hash crypté des mots de passe de vos utilisateurs pour les enregistrer de manière sécurisée dans la base de données.***

Dans le chapitre suivant, nous implémenterons notre fonction login pour vérifier les informations d'identification des utilisateurs pour leur permettre de se connecter.

***Vérifiez les informations d'identification d'un utilisateur***

*Implémentez la fonction login*

Maintenant que nous pouvons créer des utilisateurs dans la base de données, il nous faut une méthode permettant de vérifier si un utilisateur qui tente de se connecter dispose d'identifiants valides. Implémentons donc notre fonction login :

**exports.login = (req, res, next) => {**

**User.findOne({ email: req.body.email })**

**.then(user => {**

**if (!user) {**

**return res.status(401).json({ message: 'Paire login/mot de passe incorrecte'});**

**}**

**bcrypt.compare(req.body.password, user.password)**

**.then(valid => {**

**if (!valid) {**

**return res.status(401).json({ message: 'Paire login/mot de passe incorrecte' });**

**}**

**res.status(200).json({**

**userId: user.\_id,**

**token: 'TOKEN'**

**});**

**})**

**.catch(error => res.status(500).json({ error }));**

**})**

**.catch(error => res.status(500).json({ error }));**

**};**

Dans cette fonction :

Nous utilisons notre modèle Mongoose pour vérifier que l'e-mail entré par l'utilisateur correspond à un utilisateur existant de la base de données :

Dans le cas contraire, nous renvoyons une erreur 401 Unauthorized.

Si l'e-mail correspond à un utilisateur existant, nous continuons.

Nous utilisons la fonction compare de bcrypt pour comparer le mot de passe entré par l'utilisateur avec le hash enregistré dans la base de données :

S'ils ne correspondent pas, nous renvoyons une erreur401 Unauthorized avec le même message que lorsque l’utilisateur n’a pas été trouvé, afin de ne pas laisser quelqu’un vérifier si une autre personne est inscrite sur notre site.

S'ils correspondent, les informations d'identification de notre utilisateur sont valides. ***Dans ce cas, nous renvoyons une réponse 200 contenant l'ID utilisateur et un token. Ce token est une chaîne générique pour l'instant, mais nous allons le modifier et le crypter dans le prochain chapitre.***

***Avant d'utiliser la section « Partie 3 » de l'application front-end, supprimez tous les articles en vente de la section « Parties 1+2 ». Autrement, vous ne pourrez plus les modifier. En effet, comme ils ont été créés à l'aide d'un ID utilisateur générique, ils ne sont pas modifiables par des utilisateurs créés à partir de maintenant.***

La méthode compare de bcrypt compare un string avec un hash pour, par exemple, vérifier si un mot de passe entré par l'utilisateur correspond à un hash sécurisé enregistré en base de données. Cela montre que même bcrypt ne peut pas décrypter ses propres hashs.

Dans le chapitre suivant, vous découvrirez l'authentification par token, son rôle, son mode de fonctionnement et comment nous l'appliquerons dans notre nouvelle application pour sécuriser correctement notre API.

***Créez des tokens d'authentification***

*Créez des tokens d'authentification*

Les tokens d'authentification permettent aux utilisateurs de se connecter une seule fois à leur compte. Au moment de se connecter, ils recevront leur token et le renverront automatiquement à chaque requête par la suite. Ceci permettra au back-end de vérifier que la requête est authentifiée.

Pour pouvoir créer et vérifier les tokens d'authentification, il nous faudra un nouveau package :

**npm install jsonwebtoken**

Nous l'importerons ensuite dans notre contrôleur utilisateur :

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

Enfin, nous l'utiliserons dans notre fonction login :

**exports.login = (req, res, next) => {**

**User.findOne({ email: req.body.email })**

**.then(user => {**

**if (!user) {**

**return res.status(401).json({ error: 'Utilisateur non trouvé !' });**

**}**

**bcrypt.compare(req.body.password, user.password)**

**.then(valid => {**

**if (!valid) {**

**return res.status(401).json({ error: 'Mot de passe incorrect !' });**

**}**

**res.status(200).json({**

**userId: user.\_id,**

**token: jwt.sign(**

**{ userId: user.\_id },**

**'RANDOM\_TOKEN\_SECRET',**

**{ expiresIn: '24h' }**

**)**

**});**

**})**

**.catch(error => res.status(500).json({ error }));**

**})**

**.catch(error => res.status(500).json({ error }));**

**};**

Dans le code ci-dessus :

Nous utilisons la fonction sign de jsonwebtoken pour chiffrer un nouveau token.

Ce token contient l'ID de l'utilisateur en tant que payload (les données encodées dans le token).

Nous utilisons une chaîne secrète de développement temporaire RANDOM\_SECRET\_KEY pour crypter notre token (à remplacer par une chaîne aléatoire beaucoup plus longue pour la production). Puisque cette chaîne sert de clé pour le chiffrement et le déchiffrement du token, elle doit être difficile à deviner, sinon n’importe qui pourrait générer un token en se faisant passer pour notre serveur.

***Nous définissons la durée de validité du token à 24 heures. L'utilisateur devra donc se reconnecter au bout de 24 heures.***

Nous renvoyons le token au front-end avec notre réponse.

Vous pouvez désormais utiliser l'onglet « Réseau » de Chrome DevTools pour vérifier qu’une fois l’utilisateur connecté, chaque requête provenant du front-end contient bien un en-tête « Authorization » avec le mot-clé « Bearer » et une longue chaîne chiffrée. Il s'agit de notre token !

Les JSON web tokens sont des tokens chiffrés qui peuvent être utilisés pour l'autorisation.

La méthode sign() du package jsonwebtoken utilise une clé secrète pour chiffrer un token qui peut contenir un payload personnalisé et avoir une validité limitée.

Dans le chapitre suivant, nous créerons un élément de middleware pour vérifier ce token et son contenu, afin de nous assurer que seules les requêtes autorisées ont accès aux routes à protéger.

***Configurez le middleware d'authentification***

*Implémentez le middleware d'authentification*

Nous allons à présent créer le middleware qui va vérifier que l’utilisateur est bien connecté et transmettre les informations de connexion aux différentes méthodes qui vont gérer les requêtes.

Créez un dossier middleware et un fichier auth.js à l'intérieur :

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

**module.exports = (req, res, next) => {**

**try {**

**const token = req.headers.authorization.split(' ')[1];**

**const decodedToken = jwt.verify(token, 'RANDOM\_TOKEN\_SECRET');**

**const userId = decodedToken.userId;**

**req.auth = {**

**userId: userId**

**};**

**next();**

**} catch(error) {**

**res.status(401).json({ error });**

**}**

**};**

Dans ce middleware :

Étant donné que de nombreux problèmes peuvent se produire, nous insérons tout à l'intérieur d'un bloc try...catch.

Nous extrayons le token du header Authorization de la requête entrante. N'oubliez pas qu'il contiendra également le mot-clé Bearer. Nous utilisons donc la fonction split pour tout récupérer après l'espace dans le header. Les erreurs générées ici s'afficheront dans le bloc catch.

Nous utilisons ensuite la fonction verify pour décoder notre token. Si celui-ci n'est pas valide, une erreur sera générée.

Nous extrayons l'ID utilisateur de notre token et le rajoutons à l’objet Request afin que nos différentes routes puissent l’exploiter.

Dans le cas contraire, tout fonctionne et notre utilisateur est authentifié. Nous passons à l'exécution à l'aide de la fonction next().

Maintenant, nous devons appliquer ce middleware à nos routes stuff, qui sont celles à protéger. Dans notre routeur stuff, nous importons notre middleware et le passons comme argument aux routes à protéger :

**const express = require('express');**

**const router = express.Router();**

**const auth = require('../middleware/auth');**

**const stuffCtrl = require('../controllers/stuff');**

**router.get('/', auth, stuffCtrl.getAllStuff);**

**router.post('/', auth, stuffCtrl.createThing);**

**router.get('/:id', auth, stuffCtrl.getOneThing);**

**router.put('/:id', auth, stuffCtrl.modifyThing);**

**router.delete('/:id', auth, stuffCtrl.deleteThing);**

**module.exports = router;**

Désormais, à partir du front-end, vous devriez être capable de vous connecter et d'utiliser normalement l'application. Pour vérifier que les requêtes non autorisées ne fonctionnent pas, vous pouvez utiliser une application (telle que Postman) pour passer une demande sans en-tête Authorization. L'API refusera l'accès et renverra une réponse 401.

***Félicitations ! Votre API implémente à présent l'authentification par token et est correctement sécurisée.***

La méthode verify() du package jsonwebtoken permet de vérifier la validité d'un token (sur une requête entrante, par exemple).

Ajoutez bien votre middleware d'authentification dans le bon ordre sur les bonnes routes.

Attention aux failles de sécurité !

Qu'avez-vous appris dans cette partie du cours ?

Vous avez ajouté un modèle de données User afin de stocker les informations utilisateur dans votre base de données.

Vous avez implémenté le cryptage de mot de passe sécurisé afin de stocker en toute sécurité les mots de passe utilisateur.

Vous avez créé et envoyé des tokens au front-end pour authentifier les requêtes.

Vous avez ajouté le middleware d'authentification pour sécuriser les routes dans votre API. De cette façon, seules les requêtes authentifiées seront gérées.

Dans la dernière partie de ce cours, vous en apprendrez plus sur la gestion des fichiers :

comment capturer les fichiers provenant du front-end ;

comment les enregistrer sur votre serveur ;

comment les supprimer lorsqu'ils ne sont plus nécessaires.

**3 éme PARTIE:**

***Acceptez les fichiers entrants avec multer***

Dans cette dernière partie du cours, nous allons implémenter des téléchargements de fichiers pour que les utilisateurs puissent télécharger des images d'articles à vendre. Pour ce faire, nous utiliserons multer , un package qui nous permet de gérer les fichiers entrants dans les requêtes HTTP. Commençons par installer multer et par créer un élément de middleware pour gérer ces fichiers entrants.

*Configurez le middleware de gestion des fichiers*

Nous devrons d'abord installer multer dans notre projet :

**npm install multer**

Les images seront enregistrées dans un sous-dossier appelé images . Créez donc ce sous-dossier dans votre dossier backend .

Vous pouvez maintenant créer un middleware dans notre dossier middleware appelé multer-config.js :

**const multer = require('multer');**

**const MIME\_TYPES = {**

**'image/jpg': 'jpg',**

**'image/jpeg': 'jpg',**

**'image/png': 'png'**

**};**

**const storage = multer.diskStorage({**

**destination: (req, file, callback) => {**

**callback(null, 'images');**

**},**

**filename: (req, file, callback) => {**

**const name = file.originalname.split(' ').join('\_');**

**const extension = MIME\_TYPES[file.mimetype];**

**callback(null, name + Date.now() + '.' + extension);**

**}**

**});**

**module.exports = multer({storage: storage}).single('image');**

Dans ce middleware :

Nous créons une constante storage , à passer à multer comme configuration, qui contient la logique nécessaire pour indiquer à multer où enregistrer les fichiers entrants :

la fonction destination indique à multer d'enregistrer les fichiers dans le dossier images ;

la fonction filename indique à multer d'utiliser le nom d'origine, de remplacer les espaces par des underscores et d'ajouter un timestamp Date.now() comme nom de fichier. Elle utilise ensuite la constante dictionnaire de type MIME pour résoudre l'extension de fichier appropriée.

Nous exportons ensuite l'élément multer entièrement configuré, lui passons notre constante storage et lui indiquons que nous gérerons uniquement les téléchargements de fichiers image.

***multer est un package de gestion de fichiers.***

Sa méthode diskStorage() configure le chemin et le nom de fichier pour les fichiers entrants.

Sa méthode single() crée un middleware qui capture les fichiers d'un certain type (passé en argument), et les enregistre au système de fichiers du serveur à l'aide du storage configuré.

Avant de pouvoir appliquer notre middleware à nos routes stuff , nous devrons les modifier quelque peu, car la structure des données entrantes n'est pas tout à fait la même avec des fichiers et des données JSON. Voyons ça dans le chapitre suivant !

***Modifiez les routes pour prendre en compte les fichiers***

Pour que notre middleware de téléchargement de fichiers fonctionne sur nos routes, nous devrons modifier ces dernières, car le format d'une requête contenant un fichier du front-end est différent.

Nous en profiterons aussi pour implémenter les règles de sécurité. Elles nous permettront de vérifier que la personne souhaitant modifier l’objet est celle qui l’a créé ou de récupérer l’ID de l’utilisateur à partir du token d’authentification.

*Modifiez la route POST*

Tout d'abord, ajoutons notre middleware multer à notre route POST dans notre routeur stuff :

**const express = require('express');**

**const router = express.Router();**

**const auth = require('../middleware/auth');**

**const multer = require('../middleware/multer-config');**

**const stuffCtrl = require('../controllers/stuff');**

**router.get('/', auth, stuffCtrl.getAllStuff);**

**router.post('/', auth, multer, stuffCtrl.createThing);**

**router.get('/:id', auth, stuffCtrl.getOneThing);**

**router.put('/:id', auth, stuffCtrl.modifyThing);**

**router.delete('/:id', auth, stuffCtrl.deleteThing);**

**module.exports = router;**

L'ordre des middlewares est important ! Si nous plaçons multer avant le middleware d'authentification, même les images des requêtes non authentifiées seront enregistrées dans le serveur. Veillez à placer multer après auth !

Pour gérer correctement la nouvelle requête entrante, nous devons mettre à jour notre contrôleur :

**exports.createThing = (req, res, next) => {**

**const thingObject = JSON.parse(req.body.thing);**

**delete thingObject.\_id;**

**delete thingObject.\_userId;**

**const thing = new Thing({**

**...thingObject,**

**userId: req.auth.userId,**

**imageUrl: `${req.protocol}://${req.get('host')}/images/${req.file.filename}`**

**});**

**thing.save()**

**.then(() => { res.status(201).json({message: 'Objet enregistré !'})})**

**.catch(error => { res.status(400).json( { error })})**

**};**

Que fait le code ci-dessus ?

Pour ajouter un fichier à la requête, le front-end doit envoyer les données de la requête sous la forme form-data et non sous forme de JSON. Le corps de la requête contient une chaîne thing, qui est simplement un objetThing converti en chaîne. Nous devons donc l'analyser à l'aide de JSON.parse() pour obtenir un objet utilisable.

Nous supprimons le champ\_userId de la requête envoyée par le client car nous ne devons pas lui faire confiance (rien ne l’empêcherait de nous passer le userId d’une autre personne). Nous le remplaçons en base de données par le \_userId extrait du token par le middleware d’authentification.

Nous devons également résoudre l'URL complète de notre image, car req.file.filename ne contient que le segment filename. Nous utilisons req.protocol pour obtenir le premier segment (dans notre cas 'http'). Nous ajoutons '://', puis utilisons req.get('host') pour résoudre l'hôte du serveur (ici, 'localhost:3000'). Nous ajoutons finalement '/images/' et le nom de fichier pour compléter notre URL.

Pensez à utiliser la section Partie 4 ! Si vous enregistrez le contrôleur en l'état et testez l'application, vous constaterez que presque tout fonctionne. Le seul problème que nous avons est l'erreur 404 que nous obtenons lors de la tentative d'extraction de l'image, alors que notre URL semble correcte. Que se passe-t-il donc ici ?

En fait, nous effectuons une demande GET vers [http://localhost:3000/images/ HYPERLINK "http://localhost:3000/images/%3Cimage-name%3E.jpg"< HYPERLINK "http://localhost:3000/images/%3Cimage-name%3E.jpg"image-name HYPERLINK "http://localhost:3000/images/%3Cimage-name%3E.jpg"> HYPERLINK "http://localhost:3000/images/%3Cimage-name%3E.jpg".jpg](http://localhost:3000/images/%3Cimage-name%3E.jpg). Cela semble simple, mais n'oubliez pas que notre application s'exécute sur localhost:3000 et nous ne lui avons pas indiqué comment répondre aux requêtes transmises à cette route : elle renvoie donc une erreur 404. Pour remédier à cela, nous devons indiquer à notre app.js comment traiter les requêtes vers la route /image, en rendant notre dossier images statique.

Il nous faudra une nouvelle importation dans app.js pour accéder au path de notre serveur :

**const path = require('path');**

De plus, nous ajoutons le gestionnaire de routage suivant juste au-dessus de nos routes actuelles :

**app.use('/images', express.static(path.join(\_\_dirname, 'images')));**

Cela indique à Express qu'il faut gérer la ressource images de manière statique (un sous-répertoire de notre répertoire de base, \_\_dirname) à chaque fois qu'elle reçoit une requête vers la route /images. Enregistrez et actualisez l'application dans le navigateur. Désormais, tout devrait fonctionner correctement. Et maintenant, occupons-nous de la route PUT !

*Modifiez la route PUT*

La modification de notre route PUT est sensiblement plus compliquée, car nous devons prendre en compte deux possibilités : l'utilisateur a mis à jour l'image ou pas. Dans le premier cas, nous recevrons l'élément form-data et le fichier. Dans le second cas, nous recevrons uniquement les données JSON.

Il faut aussi nous assurer que la personne demandant la modification de l’objet est la propriétaire de celui-ci.

Tout d'abord, ajoutons multer comme middleware à notre route PUT :

**const express = require('express');**

**const router = express.Router();**

**const auth = require('../middleware/auth');**

**const multer = require('../middleware/multer-config');**

**const stuffCtrl = require('../controllers/stuff');**

**router.get('/', auth, stuffCtrl.getAllStuff);**

**router.post('/', auth, multer, stuffCtrl.createThing);**

**router.get('/:id', auth, stuffCtrl.getOneThing);**

**router.put('/:id', auth, multer, stuffCtrl.modifyThing);**

**router.delete('/:id', auth, stuffCtrl.deleteThing);**

**module.exports = router;**

**À présent, nous devons modifier notre fonction modifyThing() pour voir si nous avons reçu ou non un nouveau fichier et répondre en conséquence :**

**exports.modifyThing = (req, res, next) => {**

**const thingObject = req.file ? {**

**...JSON.parse(req.body.thing),**

**imageUrl: `${req.protocol}://${req.get('host')}/images/${req.file.filename}`**

**} : { ...req.body };**

**delete thingObject.\_userId;**

**Thing.findOne({\_id: req.params.id})**

**.then((thing) => {**

**if (thing.userId != req.auth.userId) {**

**res.status(401).json({ message : 'Not authorized'});**

**} else {**

**Thing.updateOne({ \_id: req.params.id}, { ...thingObject, \_id: req.params.id})**

**.then(() => res.status(200).json({message : 'Objet modifié!'}))**

**.catch(error => res.status(401).json({ error }));**

**}**

**})**

**.catch((error) => {**

**res.status(400).json({ error });**

**});**

**};**

Dans cette version modifiée de la fonction, on crée un objet thingObject qui regarde si req.file existe ou non. S'il existe, on traite la nouvelle image ; s'il n'existe pas, on traite simplement l'objet entrant. On crée ensuite une instance Thing à partir de thingObject, puis on effectue la modification. Nous avons auparavant, comme pour la route POST, supprimé le champ \_userId envoyé par le client afin d’éviter de changer son propriétaire et nous avons vérifié que le requérant est bien le propriétaire de l’objet.

**Félicitations ! Notre application gère correctement les téléchargements de fichiers lorsque nous mettons de nouveaux articles en vente et lorsque nous modifions les articles existants.**

JSON.parse() transforme un objet stringifié en Object JavaScript exploitable.

Vous aurez besoin dereq.protocol et de req.get('host'), connectés par '://' et suivis de req.file.filename, pour reconstruire l'URL complète du fichier enregistré.

Configurez votre serveur pour renvoyer des fichiers statiques pour une route donnée avec express.static() et path.join().

Dans le prochain chapitre, nous verrons comment développer la fonction delete du back-end !

***Développez la fonction delete du back-end***

*Modifiez la route DELETE*

Il y a deux points auxquels nous devons faire attention lors de la suppression d’un Thing :

Tout d’abord, nous devons nous assurer que la personne qui en fait la requête est bien celle qui a créé ce Thing.

Puis en ce qui concerne la gestion des fichiers dans notre back-end, il faut absolument nous assurer qu’à chaque suppression d'un Thing de la base de données, le fichier image correspondant est également supprimé.

Dans notre contrôleur stuff, il nous faut une nouvelle importation. Il s'agit du package fs de Node :

**const fs = require('fs');**

***fs signifie « file system » (soit « système de fichiers », en français). Il nous donne accès aux fonctions qui nous permettent de modifier le système de fichiers, y compris aux fonctions permettant de supprimer les fichiers.***

À présent, nous pouvons modifier notre fonction deleteThing() :

**exports.deleteThing = (req, res, next) => {**

**Thing.findOne({ \_id: req.params.id})**

**.then(thing => {**

**if (thing.userId != req.auth.userId) {**

**res.status(401).json({message: 'Not authorized'});**

**} else {**

**const filename = thing.imageUrl.split('/images/')[1];**

**fs.unlink(`images/${filename}`, () => {**

**Thing.deleteOne({\_id: req.params.id})**

**.then(() => { res.status(200).json({message: 'Objet supprimé !'})})**

**.catch(error => res.status(401).json({ error }));**

**});**

**}**

**})**

**.catch( error => {**

**res.status(500).json({ error });**

**});**

**};**

Dans cette fonction :

Nous utilisons l'ID que nous recevons comme paramètre pour accéder au Thing correspondant dans la base de données.

Nous vérifions si l’utilisateur qui a fait la requête de suppression est bien celui qui a créé le Thing.

Nous utilisons le fait de savoir que notre URL d'image contient un segment /images/ pour séparer le nom de fichier.

Nous utilisons ensuite la fonction unlink du package fs pour supprimer ce fichier, en lui passant le fichier à supprimer et le callback à exécuter une fois ce fichier supprimé.

Dans le callback, nous implémentons la logique d'origine en supprimant le Thing de la base de données.

Notre API peut désormais gérer correctement toutes les opérations CRUD contenant des fichiers, que ce soit lorsqu'un utilisateur crée un Thing, met à jour un Thing existant ou supprime un Thing !

***Le package fs expose des méthodes pour interagir avec le système de fichiers du serveur.***

***La méthode unlink() du package fs vous permet de supprimer un fichier du système de fichiers.***

Qu'avez-vous appris dans cette partie ?

Vous avez presque terminé. Bravo !

Regardons ce que vous avez appris :

Vous avez créé un serveur Node et l'avez utilisé pour servir une application Express.

***Vous avez connecté votre application à une base de données MongoDB et, à l'aide de Mongoose, vous avez créé une API RESTful permettant les opérations CRUD (Create, Read, Update and Delete, soit Créer, Lire, Modifier et Supprimer).***

***Vous avez implémenté une authentification sécurisée à base du token JWT.***

Enfin, vous avez implémenté la gestion du téléchargement de fichiers, permettant ainsi aux utilisateurs d'ajouter et de supprimer des images.

**Félicitations ! Vous avez bientôt terminé ce cours sur Node, Express et MongoDB. Je vous laisse répondre au dernier quiz pour tester vos compétences.**

La suite ? À vous de choisir ! Vous pouvez par exemple approfondir vos connaissances back-end. Vous pouvez aussi commencer l'apprentissage d'un framework ou d'une librairie front-end avancée telle que Angular, React ou Vue.

Où que vous alliez par la suite, je vous souhaite bonne chance et happy coding !