**Node.js, Mangodb, Express.js**

1. Installation de Express.js

<https://expressjs.com/en/starter/installing.html>

Dans le terminal de vs code : npm install express

Puis dans le fichier mère.js appeler l'api :

const express = require("express");

const app = express();

Entrée du port d'écoute.

const PORT = 4000;

Appel de la fonction d'écoute

app.listen(PORT, () => console.log(`Example app(express) d'écoute sur l'adresse http://localhost:${PORT}`));

Installer nodemon pour éviter de relancer le serveur :

Faire dans le terminal de vs code 🡺 npm install nodemon

Dans le fichier package.json 🡺 "start": "nodemon serveur.js"

"serveur.js est le point d'entrée du programme."

1. Installation de Mangodb

<https://docs.mongodb.com/manual/>

Installer l'exécutable et compasse.

A la racine, de l'ordinateur, ( c:\) , il faut créé un dossier "data" puis a l'intérieur un autre dossier "db".

Copier l'adresse du dossier "bin " de Mongodb, puis aller dans variable d'environnement, cliquer deux fois sur "path", puis sur nouveau et coller l'adresse du dossier "bin", puis ok.

Ouvrer le terminal :

mongod et entrer. => fait démarrer le serveur, (ne pas fermer).

Dans une nouvelle fenêtre de terminal :

mongo et entrer. => accès à la base de données.

Utilisation de mongodb :

* use nomDeDossier 🡺 Création du dossier, et entrée dans le dossier.
* db.nomDeDossier.insertOne({objet}) 🡺 Insertions de valeur en forme d'objet dans la base de donnée.
* Db.nomDeDossier.find() 🡺 ouvre les objets en cours dans le dossier
* db.nomDeDossier.insertMany({plusieurs objets}) 🡺 insertion de plusieurs objets dans la dataBase.
* OPENCLASSROOMS :

# [Passez au Full stack avec Node.js, Express et MongoDB](https://openclassrooms.com/fr/courses/6390246-passez-au-full-stack-avec-node-js-express-et-mongodb)

ode.js, Express et MongoDB

Fichier app.js contient le rooter (API) :

// middleware => serie de fonction qui envoi et reçoi des requetes

// Chaque réponses ne peuvent être envoyées qu’une fois.

const express = require("express");

const app = express();

//middlewre simple avec fonction next() pour quelles s'enchaines.

app.use((req, res, next)=>{

    console.log("Requête bien reçu !");

    next(); // fonction qui permet de passé au middleware suivant.

});

app.use((req, res, next)=>{

    res.status(201); // modifie le code de la réponse

    next();

});

app.use((req, res, next)=>{

    res.json({message: "Votre requete à bien été reçu !"});// réponse envoyée

    next();

});

app.use((req, res, next)=>{

    console.log("La réponse à été envoyée avec succès")

});

module.exports = app;

MiddleWare =>

* Le premier enregistre « Requête reçue ! » dans la console et passe l'exécution ;
* Le deuxième ajoute un code d'état 201 à la réponse et passe l'exécution ;
* Le troisième envoie la réponse JSON et passe l'exécution ;
* Le dernier élément de middleware enregistre « Réponse envoyée avec succès ! » dans la console.

Fichier server.js contient le serveur.

const http = require("http");

const app = require("./app");

/\*

\*Reponse de base

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

\*    res.end("Voilà la réponse du super server !");

\*});

\*/

app.set("port", process.env.PORT || 3000)

(port défini, la variable d’environnement ou le port 3000)

const server = http.createServer(app);

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

Après quelques améliorations pour la stabilité sur fichier server.js :

const http = require("http");

const app = require("./app");

const normalisePort = (val) => {

  const port = parseInt(val, 10);

  if (isNaN(port)) {

    return val;

  }

  if (port >= 0) {

    return port;

  }

  return false;

};

const port = normalisePort(process.env.PORT || 3000);

app.set("port", port);

const errorHandler = error =>{

    if(error.syscall !== 'listen'){

        throw error;

    }

    const adress = server.address();

    const bind = typeof adress === "string" ? "pipe" + adress : "port" + port;

    switch (error.code){

        case "EACCES":

            console.error(bind + "Besoin de privileges élever.");

            process.exit(1);

            break;

        case "EADDRINUSE":

            console.error(bind + "est déjà utilisé");

            process.exit(1);

            break;

        default:

            throw error;

    }

};

const server = http.createServer(app);

server.on("error", errorHandler);

server.on("listening", ()=>{

    const adress = server.address();

    const bind = typeof adress === "string" ? "pipe" + adress : "port" + port;

    console.log("En écoute sur : "+bind);

})

server.listen(port);

* 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.

Passer au concret sur le fichier app.js. Le GET.

// middleware => serie de fonction qui envoi et reçoi des requetes

// Chaque réponses ne peuvent être envoyées qu’une fois.

const express = require("express");

const app = express();

// MiddelWare qui sera appliqué à toutes les routes, toutes les requetes envoyé au serveur, ne contient pas de routes spécifiques avant les requêtes en paramètres.

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");

    // res.SetHeader donne acces à toutes les requetes de notre serveur.

    next();

});

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

   // (Npoint => url visée par l’application (String))

    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,  // en centimes pour eviter les agorithmes compliqué.

        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);

    // revois un statut 200(ok) puis un objet en format Json.

  });

module.exports = app;

Ces headers permettent :

* D'accéder à notre API depuis n'importe quelle origine ( '\*' ) ;
* D'ajouter les headers mentionnés aux requêtes envoyées vers notre API (Origin , X-Requested-With , etc.) ;
* 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.

Pour le POST :

Pour gérer la demande POST provenant de l'application front-end, nous devrons être capables d'extraire l'objet JSON de la demande. Il nous faudra le package body-parser. Installez-le en tant que dépendance de production à l'aide de npm :

npm install --save body-parser

Importez-le dans votre fichier app.js :

*const* bodyParser = require('body-parser');

Enfin, définissez sa fonction json comme *middleware* global pour votre application, juste après avoir défini les headers de la réponse :

app.use(bodyParser.json());

Maintenant, body-parser a analysé le corps de la demande. Au lieu de l'écrire dans un *middleware*.use() qui traiterait toutes les requêtes, nous allons l'écrire dans un.post() qui ne traitera que les requêtes de type POST :

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

*console*.log(req.body);

res.status(201).json({

message: 'Objet créé !'

});

});

Veillez à placer la route POST au-dessus du *middleware*pour les demandes GET, car la logique GET interceptera actuellement toutes les demandes envoyées à votre point de terminaison /api/stuff, étant donné qu'on ne lui a pas spécifié de verbe spécifique. Placer la route POST au-dessus interceptera les demandes POST, en les empêchant d'atteindre le *middleware* GET.

Désormais, si vous remplissez le formulaire dans l'application front-end et que vous l'envoyez, l'objet que vous venez de créer doit s'enregistrer dans votre console Node !

Code actuelle du fichier app.js :

// middleware => serie de fonction qui envoi et reçoi des requetes

// Chaques reponses ne peuvent être envoyées q'une fois.

const express = require("express");

const bParser = require("body-parser");

// permet de transformer le corps des requetes en json

const app = express();

// Ce MiddelWare qui sera appliqué à toutes les routes, toutes les

requêtes envoyé au serveur, ne

contient pas de routes spécifique avant les requetes en parametres.

app.use((req,res,next)=>{  // contre le CORS

    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");

    // res.SetHeader donne acces à toutes les requetes de notre serveur.

    next();

});

app.use(bParser.json()); // <= transformes les requetes en json utilisable

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

    console.log(req.body); // affiche le resultat après avoir créer un objet.

    res.status(201).json({

        message:"Objet créé !"

    });

});

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

   // (Npoint => url visé par l'application(String))

    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,  // en centimes pour eviter les agorithmes compliqué.

        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);

    // revoi un status 200(ok) puis un objet en format Json.

  });

module.exports = app;

Installation de mongoose : npm install –save mongoose

Mongoose est un package qui facilite les interactions avec notre base de données MongoDB grâce à des fonctions extrêmement utiles.

Installation de mongoDB :

Accédez au [site Web de MongoDB](https://www.mongodb.com/cloud/atlas) et inscrivez-vous pour obtenir un compte gratuit. Une fois que vous avez accès à votre tableau de bord, créez un cluster puis configurez-le avec l'**option AWS**et **uniquement les options gratuites** afin de pouvoir développer gratuitement.

Pendant le démarrage de votre cluster, vous pouvez accéder à l'onglet **Database Access**. D'abord, vous devrez ajouter un utilisateur disposant de la capacité de lecture et d'écriture dans n'importe quelle base de données. Choisissez le nom d'utilisateur ainsi que le mot de passe de votre choix et**notez-les**, car vous en aurez besoin pour connecter votre API à votre cluster.

Jusqu'à présent, nous n'avons pas pu faire persister de données ou rendre notre application entièrement dynamique. Mais tout cela est sur le point de changer, car nous intégrons la couche de base de données de notre serveur : MongoDB.

Bien qu'il soit possible de télécharger et d'exécuter MongoDB sur votre propre machine (reportez-vous au [site Web de MongoDB](https://www.mongodb.com/download-center?initial=true#community) pour en savoir plus), pour ce cours nous utiliserons la couche gratuite de MongoDB Atlas, la « database as a service » (base de données en tant que service).

Vous devrez également accéder à l'onglet **Network Access**, cliquer sur Add IP Adress et autoriser l'accès depuis n'importe où (Add access from Anywhere).

Une fois la création de votre cluster terminée, vous pouvez passer à la section suivante.

### Connectez votre API à votre cluster MongoDB

Depuis MongoDB Atlas, cliquez sur le bouton **Connect** et choisissez **Connect your application**. Sélectionnez bien la version la plus récente du driver Node.js, puis **Connection String Only**, et faites une copie de la chaîne de caractères retournée.

Après avoir installé mongoose :

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://Yan\_Coquoz:Finalf182319@cluster0.t1htf.mongodb.net/Thing?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.

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.

// création de la structure de la base de données (schéma)

const mongoose = require("mongoose");

const mySchema = mongoose.Schema({

  title: { type: String, require: true },

  description: { type: String, require: true },

  imageUrl: { type: String, require: true },

  userId: { type: String, require: true },

  price: { type: Number, require: true },

});

module.exports = mongoose.model("thing", mySchema);

* 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.

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

const Thing = require('./models/things');

Remplacement de la fonction app.post ( ):

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

    delete req.body.\_id;

    const thing = new Thing({

      ...req.body //L'opérateur spread ... est utilisé pour faire une copie de tous les éléments de req.body

    });

    thing.save()

    .then(()=> res.status(201).json({message: ">Objet enregisté !"}))

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

});

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

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.

Remplacement de la méthode app.use( ) :

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

    Thing.find()

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

    //Reccuperation de tous les things de la base

    .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.

On modifie le app.use en app.get., puis on ajoute un autre middleware app.get (pour l’id) sous le app.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(400).json({error}));

});

* 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.

Ajout maintenant des modifications et des suppressions :

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

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

  .then(()=> res.status(201).json({message: "Objet modifié !"}))

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

  //({objetàModifier}, {... pour reccupper le thing dans le corps de la requete, modificationsObjets})

});

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.

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

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

  .then(()=> res.status(201).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 ou d'échec 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 !

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 :

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 :

Fichier stuff.js

const express = require("express");

const router = express.Router(); // remplace app

const Thing = require('../models/things');// chemin local

// permet de transformer le corps des requetes en json

//! post

//Attention le chemin change

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

    delete req.body.\_id;

    // \_id créér automatiquement par mongodb

    const thing = new Thing({

      ...req.body //L'opérateur spread ... est utilisé pour faire une copie de tous les éléments de req.body

    });

    thing.save()

    .then(()=> res.status(201).json({message: "Objet enregisté !"}))

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

});

//! PUT

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

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

  .then(()=> res.status(201).json({message: "Objet modifié !"}))

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

  //({objetàModifier}, {... pour reccupper le thing dans le corps de la requete, modificationsObjets})

});

//! delete

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

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

  .then(()=> res.status(201).json({message: "Objet supprimé !"}))

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

});

//! get

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}));// 404=>obj non trouver

});

router.get('/', (req, res, next) => {

    Thing.find()

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

    //Reccuperation de tous les things de la base

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

  });

module.exports = router;

Dans le fichier app.js, on importe le router

const stuffRoutes = require("./routes/stuff");// importation de routes

puis on indique le nouveau chemin des routes que l’on a déplacé, dans une variable.

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

app.use("/api/stuff", stuffRoutes); Ancienne adresses des middlewares importer, variable du chemin

Pour rendre notre structure encore plus modulaire, 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.

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

Voici le router final :

const express = require("express");

const router = express.Router(); //remplace app

const Thing = require('../models/things');// chemin local

// permet de transformer le corps des requetes en json

// Chemin des controllers

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

router.post("/", stuffCtrl.createThings );

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

router.delete("/:id", stuffCtrl.delThing );

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

router.get('/', stuffCtrl.getThing );

module.exports = router;

Voici le fichier stuff.js du controller :

const Thing = require('../models/things');

// le post

exports.createThings = (req, res, next)=>{

    delete req.body.\_id;

    const thing = new Thing({

      ...req.body

    });

    thing.save()

    .then(()=> res.status(201).json({message: ">Objet enregisté !"}))

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

// le put

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

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

    .then(()=> res.status(201).json({message: ">Objet modifié !"}))

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

  };

  // delete

  exports.delThing = (req, res, next)=>{

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

    .then(()=> res.status(201).json({message: ">Objet supprimé !"}))

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

  };

  //get id

  exports.getIdThing = (req,res,next)=>{

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

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

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

  };

  // get

  exports.getThing = (req, res, next) => {

    Thing.find()

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

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

  };

Comme vous le voyez, cela facilite la compréhension de notre fichier de routeur. Il est évident quelles routes sont disponibles à quels points de terminaison, 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.

**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 hash 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 hash correspondent. Le package bcrypt permet d'indiquer si les deux hash 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.

Crée un modèle de donnée :

Pour s'assurer que deux utilisateurs ne peuvent 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 pré-valider les informations avant de les enregistrer :

npm install --save mongoose-unique-validator

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

Fichier user.js :

const  mongoose  = require("mongoose");

const uniqueValidator = require("mongoose-unique-validator");

// importation de validator

const userSchema = mangoose.Schema(

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

    { password : String , required : true}

);

userSchema.plugin(uniqueValidator);

// permet l'inscription d'un user avec un email unique

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

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

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.

### 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 :

// signup => pour enregistrer un nouvel utilisateur

exports.signup = (req, res, next)=>{

};

// login => connecter des utilisateurs existant

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'); // <= router

const router = express.Router();

const userCtrl = require('../controllers/user'); // association des contrôleurs au différentes routes.

// middelware

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

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

// routes post, car le frontend envoi aussi des infos (email, mot de passe)

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 :

Ajouter dans le fichier app.js

const userRoutes = require('./routes/user'); // importation des routes users

puis,

app.use('/api/auth',userRoutes); // racine des routes liée à l’autentification

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 .

npm install --save bcrypt

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

const User = require("../models/user");

const bcrypt = require("bcrypt"); // cryptage

exports.signup = (req, res, next)=>{

    bcrypt.hash(req.body.password, 10) // cryptage asynchrone (+temps).

    // (MDP du corps de la requête, nbr de fois exécution du hash)

.then(hash => { // Hash du MDP à enregistrer dans un nouveau user

            const user = new User({ //nouveau user enregistrer dans la DB

                email: req.body.email, //email du corps de la requete

                password: hash // hash en param

            });

            user.save() // enregistrement de tous le user dans la DB

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

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

        })

        .catch(error => res.status(500).json({error}));// 500 => erreur serveur

};

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](https://www.npmjs.com/package/bcrypt) 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 ;

### 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})//pour trouver 1 utilisateur de la DB

    .then(user => {

        if(!user){ //si on à pas trouver de user

            return res.status(401).json({message : "Utilisateur non trouvé !"}); //(401)=> non autorisé

        }

        // Utilisation de bcrypt, pour comparer le user qui tente de se connecter au user de la DB

        bcrypt.compare(req.body.password, user.password)//comparaison(requete,DB)

        .then(valid =>{

            if(!valid){ // si la comparaison des MDP n’est pas bonne

                return res.status(401).json({message : "Mot de Passe incorrect !"});

            }

            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 debcrypt 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 erreur 401 Unauthorized et un message « Mot de passe incorrect ! » ;
  + 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 ». Sinon, 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.

### Créez des tokens d'authentification

Les tokens d'authentification permettent aux utilisateurs de ne se connecter qu'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 :

npm install --save jsonwebtoken

Nous l'importerons ensuite dans notre contrôleur utilisateur :

const jwt = require('jsonwebtoken');

Puis dans la fonction login() au niveau du token :

Token vérifier à chaque requête du frontend.

res.status(200).json({

                userId : user.\_id,

                token : jwt.sign( //le payload => donnée que l’on veut encodé

                    { userId : user.\_id },// id utilisateur

                    "RANDOM\_TOKEN\_SECRET",// clé de cryptage

                    {expiresIn : "24h"} // validité du token

                )

            })

Dans le code ci-dessus :

* Nous utilisons la fonction sign de jsonwebtoken pour encoder 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 encoder notre *token* (à remplacer par une chaîne aléatoire beaucoup plus longue pour la production) ;
* 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 que, une fois connecté, chaque requête provenant du front-end contient bien un en-tête « Authorization » avec le mot-clé « Bearer » et une longue chaîne encodée. Il s'agit de notre *token* !

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 protégera les routes sélectionnées et vérifier que l'utilisateur est authentifié avant d'autoriser l'envoi de ses requêtes.

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

const jwt = require('jsonwebtoken');

module.exports = (res, req, next)=>{

    try {

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

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

        const userId = decodedToken.userId;

        if (req.body.userId && req.body.userId !== userId) {

            throw "User id non valable ! ";

        }else{

            next()

        }

    } catch (error) {

        res.status(401).json({error : error | "Requête non authentifiée ! "});

    }

};

Dans ce middleware :

* Etant 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 récupérer tout 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* ;
* Si la demande contient un ID utilisateur, nous le comparons à celui extrait du *token*. S'ils sont différents, nous générons une erreur ;
* Dans le cas contraire, tout fonctionne et notre utilisateur est authentifié. Nous passons 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 :

Dans le dossier des routes , stuff.js

const express = require("express");

const router = express.Router(); //remplace app

// const Thing = require("../models/things"); // chemin local

// permet de transformer le corps des requetes en json

// Chemin des controllers

const stuffCtrl = require("../controllers/stuff");

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

router.get("/", auth , stuffCtrl.getAllThings);

router.post("/", auth , stuffCtrl.createThing);

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

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

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

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

## 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.

const multer = require ('multer');

const MINE\_TYPE ={

    'image/jpg' : 'jpg',

    'image/jpeg': 'jpg',

    'image/png' : 'jpg',

}

const storage = multer.diskStorage({

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

        callback(null, "images");// (pas d'erreur, dossier de reception)

    },

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

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

        const extension = MINE\_TYPE[file.mimetype];

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

    }

})

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

Nous devrons d'abord installer multer dans notre projet :

npm install --save 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 :

Dans ce middleware :

1. 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 ;
2. 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.

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.

## Modifiez les routes pour prendre en compte les fichiers

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

Modifier la route POST :

// Les routes

const express = require("express");

const router = express.Router(); //remplace app

// const Thing = require("../models/things"); // chemin local

// permet de transformer le corps des requetes en json

// Chemin des controllers

const stuffCtrl = require("../controllers/stuff");

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

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

router.get("/", auth , stuffCtrl.getAllThings);

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

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

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

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

module.exports = router;

L'ordre des middlewares est important ! Si nous devons placer 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 :

// le post

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

  const thingObject = json.parse(req.body.thing);

  delete thingObject.\_id;

  const thing = new Thing({

    ...thingObject,

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

  });

  thing

    .save()

    .then(() => res.status(201).json({ message: "Objet enregisté !" }))

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

};

* 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 objet Thing converti en chaîne. Nous devons donc l'analyser à l'aide de JSON.parse() pour obtenir un objet utilisable.
* 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/<image-name>.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 !

Modifier 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.

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

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

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

// le put

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};

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

    .then(() => res.status(201).json({ message: "Objet modifié !" }))

    .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.

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

### Modifiez la route DELETE

n ce qui concerne la gestion des fichiers dans notre back-end, il faut absolument nous assurer que, 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() :

// delete

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

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

    .then((thing) => {

      const fileName = thing.imageUrl.split("/images/")[1];

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

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

          .then(() => res.status(201).json({ message: "Objet supprimé !" }))

          .catch((error) => res.status(400).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 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 : lorsqu'un utilisateur crée un Thing , met à jour un Thing existant ou supprime un Thing !

### Récapitulons...

Vous avez presque terminé. Bravo !

Regardons ce que l’on a 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 — 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, choisir d'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**!

**---------------------------------------------------**

**Si une requête entrante contient un fichier ainsi que d'autres données, notre fonctionbodyParser.json() transformera automatiquement les autres données en JSON exploitable.**

*Faux. Les données entrantes sont sous la forme de form-data . Il faut donc utiliser JSON.parse() pour transformer des chaînes de caractères en JSON exploitable.*