**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.