SACCHETTO Vladimir BTS SIO

2.1 PHP - Concevoir et développer solution applicative

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

DOCUMENT DE CAS D'UTILISATION

SOMMAIRE

Configurez votre environnement de développement

Installer Node

Notre environnement se base sur une machine virtuelle Linux. Nous allons donc installer **NodeJs** ainsi que **Npm** avec les commandes suivantes :

```
curl -fsSL https://deb.nodesource.com/setup_18.x | sudo -E bash -

sudo apt-get install -y nodejs

sudo npm install -g npm@latest
```

Cloner l'application front-end

Nous allons maintenant cloner le projet **Frontend** avec la commande :

```
git clone https://github.com/OpenClassrooms-Student-Center/go-fullstack-v3-fr.git frontend
```

Puis se rendre dans le dossier cloné avec `cd frontend` puis exécuter :

- `Npm install`
- `Npm run start`

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.

À partir de votre dossier `backend`, exécutez la commande de terminal `npm init`. `Le` point d'entrée doit être `server.js`

```
ubuntu@developpeur:/var/www/NodeJs/backend$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (backend)
version: (1.0.0)
description:
entry point: (index.js) server.js
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to /var/www/NodeJs/backend/package.json:

{
  "name": "backend",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}


Is this OK? (yes) yes
```

Dans le dossier `backend` nous avons maintenant un fichier `package.json`

```
ubuntu@developpeur:/var/www/NodeJs/backend$ ll
total 20
drwxrwxr-x 3 ubuntu ubuntu 4096 oct.  31 11:19 ./
drwxrwxr-x 4 ubuntu ubuntu 4096 oct.  31 11:09 ../
drwxrwxr-x 7 ubuntu ubuntu 4096 oct.  31 11:10 .git/
-rw-rw-r-- 1 ubuntu ubuntu   14 oct.  31 11:11 .gitignore
-rw-rw-r-- 1 ubuntu ubuntu  204 oct.  31 11:19 package.json
ubuntu@developpeur:/var/www/NodeJs/backend$ cat package.json
{
  "name": "backend",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Créez ensuite un fichier `server.js` à l'intérieur de votre dossier `backend` . Il contiendra votre premier serveur Node.

```
const http = require('http')

const server = http.createServer((req, res) => {
  res.end('Voilà la réponse du serveur !')
})

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

Exécuter `node server` pour le serveur

Installez nodemon

Faire `sudo npm install -g nodemon`

```
ubuntu@developpeur:/var/www/NodeJs/backend$ sudo npm install -g nodemon

changed 32 packages, and audited 33 packages in 2s

3 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Au lieu d'utiliser `node server` pour démarrer votre serveur, on peut utiliser `nodemon server`

```
ubuntu@developpeur:/var/www/NodeJs/backend$ nodemon server
[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`
```

Installez Express

On exécute la commande `npm install express`

```
^Cubuntu@developpeur:/var/www/NodeJs/backend$ npm install express

added 57 packages, and audited 58 packages in 5s

7 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

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

```
const express = require('express')

const app = express()

module.exports = app
```

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
```

Ajoutez des middlewares

Modifier le fichier `app.js`

```
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* :

- 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.

Améliorez server.js

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.

Créez une route GET

Exécutez donc `npm run start` à partir du répertoire `front-end`, accédez à http://localhost:4200 avec votre navigateur puis cliquez sur « Parties 1 + 2 ».

```
ubuntu@developpeur:/var/www/NodeJs/frontend$ npm run start

> go-fullstack-v3-fr@1.0.0 start
> npx http-server . -p 4200 -P "http://localhost:4200?"

Starting up http-server, serving .

http-server settings:
CORS: disabled
Cache: 3600 seconds
Connection Timeout: 120 seconds
Directory Listings: visible
AutoIndex: visible
Serve GZIP Files: false
Serve Brotli Files: false
Default File Extension: none

Available on:
  http://127.0.0.1:4200
  http://172.22.170.79:4200
Unhandled requests will be served from: http://localhost:4200?
Hit CTRL-C to stop the server
```

Remettre des articles en vente

Modifier le middleware de app.js

```
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 soit '/api/stuff'.

Dans ce *middleware*, nous créons un groupe d'articles avec le schéma de données spécifique requis par le front-end. Si on effectue une demande GET vers cette route (aussi appelée *endpoint*) à partir de Insomnia, on verra qu'on reçoit le groupe de `stuff` , mais que l'actualisation du navigateur ne semble pas fonctionner.

Cela est dû aux 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`

On modifie alors le fichier `app.js` en ajoutant le *middlware* `app.use` suivant avant la route 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 :

- 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.).

Maintenant les objets sont chargés sur notre page web.

Créez une route POST

Recevez des articles de l'application front-end

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 !',
  })
})
```

Configuration de la base de données

Nous allons configurer une base de données NoSQL. Voir définition de cela via la page.

Nous allons créer un compte dans le site MongoDB.

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*.

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*).

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.

On installe ensuite le package Mongoose en exécutant, à partir du dossier `backend`, la commande suivante : `npm install 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://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 !'))
```

En résumé

- 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

Mongoose peut gérer la BDD MonoDB en implémentant des schémas de données stricts, qui permettent de rendre notre application plus robuste. On commence par créer 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.

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')
```

Puis remplacer la logique de la 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

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.

En résumé

- 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.

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 :

```
// Permet de créer une route pour Modifier l'objet ou le post avec app.put et
Thing.updateOne
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 avec succes !!' }),
    )
    .catch(() => res.status(400).json({ error }))
})
```

Nous exploitons la méthode `updateOne()` dans le 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.

Suppression d'un Thing

Nous allons ajouter 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 avec succes !' }),
    )
    .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 !

En résumé

- `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.

Optimiser la structure du back-end

Configurer le routage

Nous allons créer un sous-dossier `routes` dans le dossier `backend` 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
```

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.

Nous allons alors couper tout le système de routage du fichier `app.js` dans le fichier `stuff.js`. Nous allons apporter des modifications :

- Nous allons supprimer `/api/stuff` et le remplacer par un `/`
- Nous modifions `app.` Par `router.`
- Dans le fichier `app.js` nous allons ajouter `const stuffRoutes = require('./routes/stuff')` et `app.use('/api/stuff', stuffRoutes);`
- A la fin du fichier `stuff.js` nous ajoutons `module.exports = router`

Configurez les contrôleurs

Nous allons créer un sous-dossier `controllers` dans le dossier `backend` puis dans ce nouveau dossier, un fichier `stuff.js`. Ainsi nous allons séparer la logique métier de nos routes en contrôleurs.

```
exports.createThing = (req, res, next) => {
  delete req.body._id
  const thing = new Thing({
    ...req.body,
  })
  thing
    .save()
    // Fonctions asynchrone = async function - JavaScript fonctions promise
```

```
    .then(() => res.status(201).json({ message: 'Objet enregistré !' })) //
Fonction flachée - envoie la réponse code ok 200

    .catch((error) => res.status(400).json({ error })) // Envoie l'erreur 400

}
```

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`

```
router.post('/', stuffCtrl.createThing)
```

Nous allons effectuer ces modifications pour tous les `middlewares`.

De cette manière nous allons avoir notre `router` :

```
const express = require('express')

const router = express.Router()


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


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

router.put('/:id', stuffCtrl.updateThing)

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

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

router.get('/', stuffCtrl.getAllThing)


module.exports = router
```

Et notre `contrôleur` :

```
const Thing = require('../models/Thing')


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

  delete req.body._id

  const thing = new Thing({
```

```
    ...req.body,
  })
  thing
    .save()
    // Fonctions asynchrone = async function - JavaScript fonctions promise
    .then(() => res.status(201).json({ message: 'Objet enregistré !' })) //
Fonction flachée - envoie la réponse code ok 200
    .catch((error) => res.status(400).json({ error })) // Envoie l'erreur 400
}

exports.updateThing = (req, res, next) => {
  Thing.updateOne({ _id: req.params.id }, { ...req.body, _id: req.params.id })
    .then(() =>
      res.status(200).json({ message: 'Objet modifié avec succes !!' }),
    )
    .catch(() => res.status(400).json({ error }))
}

exports.deleteThing = (req, res, next) => {
  Thing.deleteOne({ _id: req.params.id })
    .then(() =>
      res.status(200).json({ message: 'Objet supprimé avec succes !' }),
    )
    .catch((error) => res.status(400).json({ error }))
}

exports.getOneThing = (req, res, next) => {
  Thing.findOne({ _id: req.params.id })
    .then((things) => res.status(200).json(things))
```

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


exports.getAllThing = (req, res, next) => {
  Thing.find()
    .then((things) => res.status(200).json(things))
    .catch((error) => res.status(400).json({ error }))
}
```

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.

En résumé

- 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.

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

Nous allons créer un model `User` dans le dossier `models` déjà présent dans notre dossier `backend`.

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 --save mongoose-unique-validator`

Ce package une fois qu'il sera installé, nous pouvons créer notre propre modèle utilisateur :

```
const mongoose = require('mongoose')

const uniqueValidator = require('mongoose-unique-validator') // On implement
le package mongoose-unique-validator



const userSchema = mongoose.Schema({

  email: { type: String, required: true, unique: true }, // unique: true
permet de faire enregistrer l'user une fois avec la même adresse mail

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

})



userSchema.plugin(uniqueValidator) // On implemente le plugin à notre
userSchema
```

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

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.

Configurer les routes d'authentification

Nous allons créer un `contrôleur user.js` ainsi qu'un `router user.js`.

```
// Controller user.js

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

}

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

}
```

```
// Router user.js

const express = require('express')

const router = express.Router()

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


router.post('/signup', userCtrl.signup) // Route POST pour l'enregistrement

router.post('/login', userCtrl.login) // Route POST pour l'authentification


module.exports = router
```

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

Enregistrons notre routeur dans notre application. Pour ce faire, importer le routeur :

```
const userRoutes = require('./routes/user')
```

Puis on l'enregistre :

```
app.use('/api/auth', userRoutes)
```

Créez des utilisateurs

Afin de créer un user nous allons devoir installer le package `bcrypt` afin d'hacher les mots de passe. On exécute la commande `npm install bcrypt`

Nous allons implémenter notre contrôleur avec la méthode `.signup`

```
const bcrypt = require('bcrypt')


const User = require('../models/User')


exports.signup = (req, res, next) => {
  // Methode asynchrone pour crypter le mot de passe de l'User à la création du compte
  bcrypt
    .hash(req.body.password, 10)
    .then((hash) => {
      const user = new User({
        email: req.body.email,
        pssword: hash,
      })
      user
        .save()
        .then(() => res.status(201).json({ message: 'User créé !' }))
        .catch((error) => res.status(400).json({ error }))
    })
    .catch((error) => res.status(500).json({ error }))
}
exports.login = (req, res, next) => {}
```

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.

Implémentez la fonction login

Nous allons maintenant implémenter notre fonction `.login` :

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

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

    .then((user) => {

      if (user === null) {

        res.status(401).json({ message: 'Paire identifiant/mot de passe
incorrecte' })

      } else {

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

              .then((valid) => {

                  if (!valid) {

                    res.status(401).json({message: 'Paire identifiant/mot de
passe incorrecte',

                      })

                  } else {

                      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 erreur `401 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.

Créer des tokens d'authentification

Dans notre contrôleur notre fonction login retourne une chaine de caractère aléatoire à chaque authentification. Cela est le TOKEN que nous avons défini. Nous voulons vérifier cette requête envoyée par le frontend à chaque envoi du frontend pour vérifier si la requête est correspondante à l'authentification.

Nous allons installer un package avec la commande `npm install --save jsonwebtoken`. Ce package permet de créer des tokens et de les vérifier.

Après installation nous l'importerons dans notre contrôleur utilisateur :

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

Dans notre code au lieu d'envoyer une simple chaine de caractères, on appelle une fonction de jsonwebtoken `jwt.sign` qui prend des arguments

```
token: jwt.sign(
                { userId: user._id },
                'RANDOM_TOKEN_SECRET', {
                expiresIn: '24h'}),
```

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.

En résumé

- 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.

Implémentez le *middleware* d'authentification

Créer 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
        }
    } catch (erreur) {
        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 auth = require(' ../middleware/auth')
const router = express.Router()

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

router.post('/', auth, stuffCtrl.createThing)
router.put('/:id', auth, stuffCtrl.updateThing)
router.delete('/:id', auth, stuffCtrl.deleteThing)
router.get('/:id', auth, stuffCtrl.getOneThing)
router.get('/', auth, stuffCtrl.getAllThing)

module.exports = router
```

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

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

Ajoutez une gestion des fichiers utilisateur sur l'application

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(numm, '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 }).single('image')
```

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.

En résumé

- `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.

Modifiez la route POST

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

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

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

```
const Thing = require('../models/Thing')

exports.getAllThing = (req, res, next) => {
  Thing.find()
    .then((things) => res.status(200).json(things))
    .catch((error) => res.status(400).json({ error }))
}

exports.createThing = (req, res, next) => {
  const thingObject = JSON.parse(req.body.thing)
  delete thingObject._id
  delete thingObject._userId // On utilise le token d'authentification
  const thing = new Thing({
    ...thingObject,
    userId: req.auth.userId,
    imageUrl: `${req.protocol}://${req.gest('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.

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

```
router.put('/:id', auth, multer, stuffCtrl.updateThing)
```

Nous devons modifier notre fonction `updateThing()` pour voir si nous avons reçu ou non un nouveau fichier et répondre en conséquence :

```
exports.updateThing = (req, res, next) => {
  const thingObject = req.file
    ? {
        ...JSON.parse(req.body.thing),
```

```
        imageUrl: `${req.protocol}://${req.gest('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: 'Non autorisé' }))
      } else {
        Thing.updateOne(
          { _id: req.params.id },
          { ...thingObject, _id: req.params.id },
        )
          .then(() => {
            res.status(201).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.

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')
```

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: 'Non autorisé' }))
      } else {
        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(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` !

En résumé

- `JSON.parse()` transforme un objet stringifié en Object JavaScript exploitable.
- Vous aurez besoin de`req.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()`.
- 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.