

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



Configurez votre environnement de développement.....	3
Installer Node.....	3
Clonez l'application front-end.....	3
En résumé.....	5
Qu'est-ce que Node ?.....	5
Initialisez votre projet.....	5
Installez nodemon.....	7
Installez Express.....	8
Exécutez l'application Express sur le serveur Node.....	8
Ajoutez des middlewares.....	9
Améliorez server.js.....	10
En résumé.....	11
Créez une route GET.....	12
Remettre des articles en vente.....	12
Erreurs de CORS.....	13
En résumé.....	14
Créez une route POST.....	14
En résumé.....	15
Qu'est-ce que MongoDB ?.....	16
Configurez MongoDB Atlas.....	16
Connectez votre API à votre cluster MongoDB.....	17
Créez un schéma Thing.....	18
Enregistrement des Things dans la base de données.....	18
En résumé.....	19
Modifiez et supprimez des données.....	19
Enregistrez et récupérez des données.....	19
Modifiez et supprimez des données.....	20
Optimisez la structure du back-end.....	21
Préparez la base de données pour les informations d'authentification.....	24
Créez des utilisateurs.....	25
Configurez les routes d'authentification.....	25
Créez des utilisateurs.....	26
Vérifiez les informations d'identification d'un utilisateur.....	27
Implémentez la fonction login.....	27
Créez des tokens d'authentification.....	28
Configurez le middleware d'authentification.....	29
Implémentez le middleware d'authentification.....	29

Configurez votre environnement de développement

Installer Node

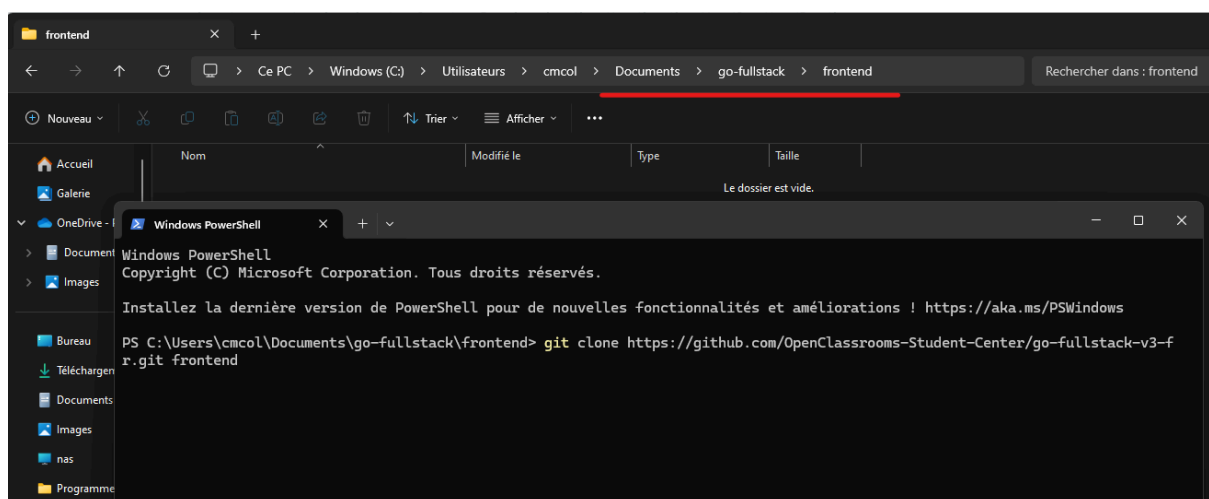
Accédez à [NodeJS.org](https://nodejs.org) pour télécharger puis installer la dernière version de Node. Cela a pour effet d'installer le *runtime* JavaScript de Node, ce qui vous permet par là même d'exécuter les serveurs Node. Cela installe également **Node Package Manager** ou **npm**, outil précieux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

Clonez l'application front-end

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

Une fois le répertoire créé, il vous faudra cloner [le code pour l'application front-end](#) dans un sous-répertoire appelé **frontend**. À partir de votre répertoire de travail, entrez la commande ci-dessous :

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



Vous pouvez ensuite procéder comme suit :

```
cd frontend
npm install
npm audit fix
npm run start
```

Cela installera toutes les dépendances requises par l'application front-end, et lancera le serveur de développement.

```
PS C:\Users\cmcol\Documents\go-fullstack\frontend\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://192.168.56.1:4200
  http://192.168.38.1:4200
  http://192.168.78.1:4200
  http://192.168.61.1:4200
  http://192.168.86.29:4200
  http://127.0.0.1:4200
Unhandled requests will be served from: http://localhost:4200?
Hit CTRL-C to stop the server
```

<http://localhost:4200>, vous devriez voir l'interface suivante (en supposant que vous ayez bien suivi les étapes ci-dessus) :

Appli front-end - Passez au Full-Stack

Parties 1+2

Partie 3

Partie 4

En résumé

- Node peut être installé à partir de [NodeJS.org](https://nodejs.org).
- L'application front-end pour ce cours peut être clonée avec `git clone`, installée avec `npm install`, et lancée avec `npm run start`

Qu'est-ce que Node ?

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

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

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

Initialisez votre projet

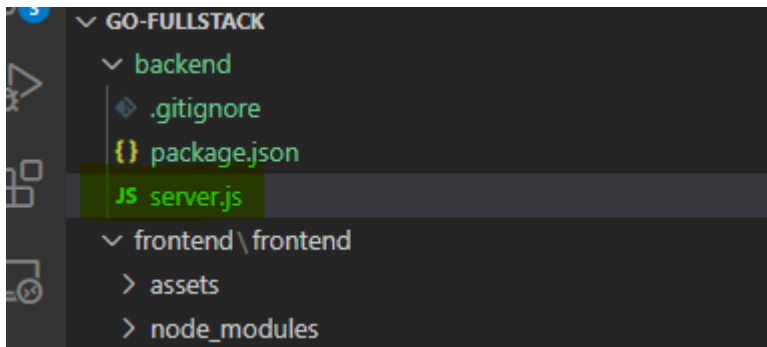
Avant de commencer, vous pouvez initialiser un dépôt Git en exécutant `git init` depuis votre dossier `backend`. N'oubliez pas de créer un fichier `.gitignore` contenant la ligne `node_modules` afin de ne pas envoyer ce dossier (qui deviendra volumineux) vers votre dépôt distant.

```
PS C:\Users\cmcol\Documents\go-fullstack\backend> git init
Initialized empty Git repository in C:/Users/cmcol/Documents/go-fullstack/backend/.git/
PS C:\Users\cmcol\Documents\go-fullstack\backend>
```

À partir de votre dossier `backend`, exécutez la commande de terminal `npm init` pour initialiser votre projet. Vous pouvez utiliser les options par défaut, ou les modifier si vous le souhaitez. Cependant, votre point d'entrée doit être `server.js`. Vous le créerez bientôt.

```
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:
```

Création du fichier `server.js` dans “backends”



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

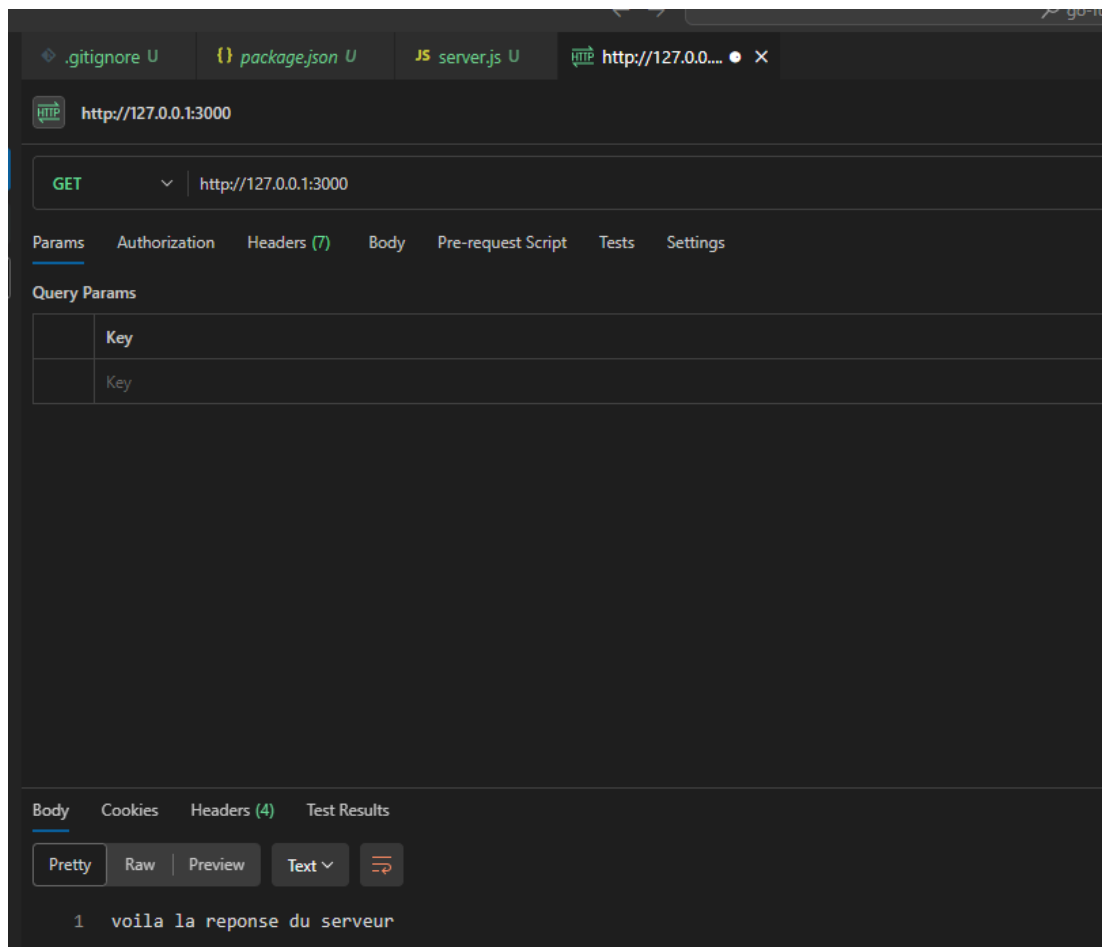
```
const http = require('http');

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

server.listen(3000);
```

Ici, vous importez le package HTTP natif de Node et l'utilisez pour créer un serveur, en passant une fonction qui sera exécutée à chaque appel effectué vers ce serveur. Cette fonction reçoit les objets `request` et `response` en tant qu'arguments. Dans cet exemple, vous utilisez la méthode `end` de la réponse pour renvoyer une réponse de type `string` à l'appelant.

On lance le server avec la commande `node server` , puis on test l'envoi d'une requête http au serveur avec postman :



En sortie on voit bien que le serveur à répondu.

Installez nodemon

Pour simplifier le développement Node, vous souhaitez peut-être installer `nodemon` . Pour ce faire, exécutez la commande suivante :

```
npm install -g nodemon
```

```
PS C:\Users\cmcol\Documents\go-fullstack\backend> nodemon server
[nodemon] 3.1.3
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting 'node server.js'
```

Désormais, au lieu d'utiliser `node server` pour démarrer votre serveur, vous pouvez utiliser `nodemon server` .

Installez Express

Pour ajouter Express à votre projet, exécutez la commande suivante à partir de votre dossier `backend` :

```
npm install express
```

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

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

const app = express();

module.exports = app;
```

Exécutez l'application Express sur le serveur Node

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

```
const http = require('http');
const app = require('./app');

app.set('port', process.env.PORT || 3000);
const server = http.createServer(app);

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

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

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

const app = express();

app.use((req, res) => {
  res.json({ message: 'Votre requête a bien été reçue !' });
});

module.exports = app;
```


Ajoutez des middlewares

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

Vous retrouvez ci-dessous l'intégralité du code :

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

const app = express();

app.use((req, res, next) => {
  console.log('Requête reçue !');
  next();
});

app.use((req, res, next) => {
  res.status(201);
  next();
});

app.use((req, res, next) => {
  res.json({ message: 'Votre requête a bien été reçue !' });
  next();
});

app.use((req, res, next) => {
  console.log('Réponse envoyée avec succès !');
});

module.exports = app;
```

Cette application Express contient quatre éléments de *middleware* :

- 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.

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

Améliorez server.js

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

```
const http = require('http');
const app = require('./app');

const normalizePort = val => {
  const port = parseInt(val, 10);

  if (isNaN(port)) {
    return val;
  }
  if (port >= 0) {
    return port;
  }
  return false;
};

const port = normalizePort(process.env.PORT || '3000');
app.set('port', port);

const errorHandler = error => {
  if (error.syscall !== 'listen') {
    throw error;
  }
  const address = server.address();
  const bind = typeof address === 'string' ? 'pipe ' + address : 'port: ' + port;
  switch (error.code) {
    case 'EACCES':
      console.error(bind + ' requires elevated privileges.');
```

```
process.exit(1);
      break;
    case 'EADDRINUSE':
      console.error(bind + ' is already in use.');
```

```
process.exit(1);
      break;
    default:
      throw error;
  }
};

const server = http.createServer(app);

server.on('error', errorHandler);
server.on('listening', () => {
```

```
const address = server.address();
const bind = typeof address === 'string' ? 'pipe ' + address : 'port '
+ port;
console.log('Listening on ' + bind);
});

server.listen(port);
```

Aperçu rapide de ce qui se passe ici :

- la fonction `normalizePort` renvoie un port valide, qu'il soit fourni sous la forme d'un numéro ou d'une chaîne ;
- la fonction `errorHandler` recherche les différentes erreurs et les gère de manière appropriée. Elle est ensuite enregistrée dans le serveur ;
- un écouteur d'évènements est également enregistré, consignait le port ou le canal nommé sur lequel le serveur s'exécute dans la console.

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

En résumé

- Le framework Express est installé et enregistré dans le `package.json` avec `npm install express` .
- Pour créer une application Express, appelez simplement la méthode `express()` .
- Un middleware est un bloc de code qui traite les requêtes et réponses de votre application.
- Ajouter la normalisation de port, la gestion d'erreur et du logging basique à votre serveur Node le rend plus constant et plus facile à déboguer.

Créez une route GET

Remettre des articles en vente

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

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

```
app.use('/api/stuff', (req, res, next) => {
  const stuff = [
    {
      _id: 'oeihfzeoi',
      title: 'Mon premier objet',
      description: 'Les infos de mon premier objet',
      imageUrl:
        'https://cdn.pixabay.com/photo/2019/06/11/18/56/camera-4267692_1280.jpg'
    },
    {
      price: 4900,
      userId: 'qsomihvqios',
    },
    {
      _id: 'oeihfzeomoihi',
      title: 'Mon deuxième objet',
      description: 'Les infos de mon deuxième objet',
      imageUrl:
        'https://cdn.pixabay.com/photo/2019/06/11/18/56/camera-4267692_1280.jpg'
    },
    {
      price: 2900,
      userId: 'qsomihvqios',
    },
  ];
  res.status(200).json(stuff);
});
```

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

Erreurs de CORS

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

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

```
app.use((req, res, next) => {  
  res.setHeader('Access-Control-Allow-Origin', '*');  
  res.setHeader('Access-Control-Allow-Headers', 'Origin,  
X-Requested-With, Content, Accept, Content-Type, Authorization');  
  res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE,  
PATCH, OPTIONS');  
  next();  
});
```

Ici, nous pouvons voir que tout marche bien :



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.).

En résumé

- La méthode `app.use()` vous permet d'attribuer un middleware à une route spécifique de votre application.
- Le CORS définit comment les serveurs et les navigateurs interagissent, en spécifiant quelles ressources peuvent être demandées de manière légitime – par défaut, les requêtes AJAX sont interdites.
- Pour permettre des requêtes cross-origin (et empêcher des erreurs CORS), des headers spécifiques de contrôle d'accès doivent être précisés pour tous vos objets de réponse.

Créez une route POST

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) => {  
  
  app.post('/api/stuff', (req, res, next) => {  
    console.log(req.body);  
    res.status(201).json({  
      message: 'Objet créé !'  
    });  
  });  
});
```

RE **VENDRE UN OBJET**

Titre

Prix (en €)

Description

URL de l'image

Valider

```
[nodemon] starting 'node server.js'
Listening on port 3000
{
  title: 'Un nouvel objet',
  description: "plein d'info",
  price: 10000,
  imageUrl: 'dedededed',
  _id: '1718027562496',
  userId: 'userID40282382'
}
```

En résumé

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

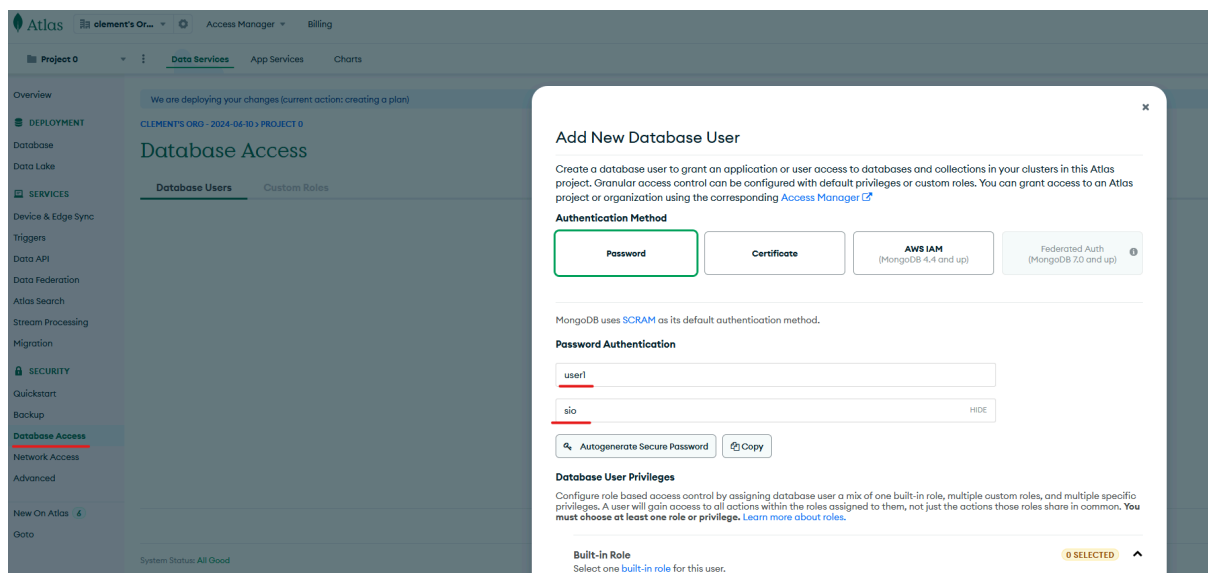
Qu'est-ce que MongoDB ?

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

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

Configurez MongoDB Atlas

Bien qu'il soit possible de télécharger et d'exécuter MongoDB sur votre propre machine (reportez-vous au [site web de MongoDB](#) 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).



Autogenerate Secure Password Copy

Database User Privileges

Configure role based access control by assigning database user a mix of one built-in role, multiple custom roles, and multiple specific privileges. A user will gain access to all actions within the roles assigned to them, not just the actions those roles share in common. **You must choose at least one role or privilege.** [Learn more about roles.](#)

Built-in Role

Select one **built-in role** for this user.

1 SELECTED

Read and write to any database

User Name	Authentication Method	MongoDB Roles
user1	SCRAM	readWriteAnyDatabase@admin

Puis se rendre dans "Network access" / "Add Ip Adresse" et cliquer sur "Allow 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.

De retour sur votre projet, installez 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 :

```
mongodb+srv://<username>:<password>@cluster0.0ucumif.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0
```

```
mongoose.connect('mongodb+srv://jimboj:<sio>@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 !'));
```

Créez un schéma Thing

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

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

const thingSchema = mongoose.Schema({
  title: { type: String, required: true },
  description: { type: String, required: true },
  imageUrl: { type: String, required: true },
  userId: { type: String, required: true },
  price: { type: Number, required: true },
});

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

Enregistrement des Things dans la base de données

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 }));
});
```

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.

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

Modifiez et supprimez des données

- `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.

Enregistrez et récupérez des données

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

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

Maintenant, remplacez la logique de votre route POST par :

```
app.post('/api/stuff', (req, res, next) => {
  delete req.body._id;
  const thing = new Thing({
    ...req.body
  });
  thing.save()
    .then(() => res.status(201).json({ message: 'Objet enregistré !' }))
    .catch(error => res.status(400).json({ error }));
});
```

```
});
```

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 }));
});
```

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 }));
});
```

Modifiez et supprimez des données

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

```
app.put('/api/stuff/:id', (req, res, next) => {
  Thing.updateOne({ _id: req.params.id }, { ...req.body, _id:
req.params.id })
    .then(() => res.status(200).json({ message: 'Objet modifié !'}))
    .catch(error => res.status(400).json({ error }));
});
```

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

```
app.delete('/api/stuff/:id', (req, res, next) => {
  Thing.deleteOne({ _id: req.params.id })
    .then(() => res.status(200).json({ message: 'Objet supprimé !'}))
    .catch(error => res.status(400).json({ error }));
});
```

Maintenant notre site est fonctionnelle nous pouvons ajouter / modifier / supprimer un objet



Optimisez la structure du back-end

Il est temps de couper toutes nos routes de `app.js` et de les coller dans notre routeur.

Veillez à remplacer toutes les occurrences de `app` par `router`, car nous enregistrons les routes dans notre routeur :

```
const express = require('express');
const router = express.Router();

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

router.post('/', (req, res, next) => {
  const thing = new Thing({
    title: req.body.title,
    description: req.body.description,
    imageUrl: req.body.imageUrl,
    price: req.body.price,
    userId: req.body.userId
  });
  thing.save().then(
    () => {
      res.status(201).json({
```

```

        message: 'Post saved successfully!'
    });
    }
).catch(
    (error) => {
        res.status(400).json({
            error: error
        });
    }
);
});

router.get('/:id', (req, res, next) => {
    Thing.findOne({
        _id: req.params.id
    }).then(
        (thing) => {
            res.status(200).json(thing);
        }
    ).catch(
        (error) => {
            res.status(404).json({
                error: error
            });
        }
    );
});

router.put('/:id', (req, res, next) => {
    const thing = new Thing({
        _id: req.params.id,
        title: req.body.title,
        description: req.body.description,
        imageUrl: req.body.imageUrl,
        price: req.body.price,
        userId: req.body.userId
    });
    Thing.updateOne({_id: req.params.id}, thing).then(
        () => {
            res.status(201).json({
                message: 'Thing updated successfully!'
            });
        }
    ).catch(
        (error) => {
            res.status(400).json({

```

```

        error: error
      });
    }
  );
});

router.delete('/:id', (req, res, next) => {
  Thing.deleteOne({_id: req.params.id}).then(
    () => {
      res.status(200).json({
        message: 'Deleted!'
      });
    }
  ).catch(
    (error) => {
      res.status(400).json({
        error: error
      });
    }
  );
});

router.get('/', (req, res, next) => {
  Thing.find().then(
    (things) => {
      res.status(200).json(things);
    }
  ).catch(
    (error) => {
      res.status(400).json({
        error: error
      });
    }
  );
});

module.exports = router;

```

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

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

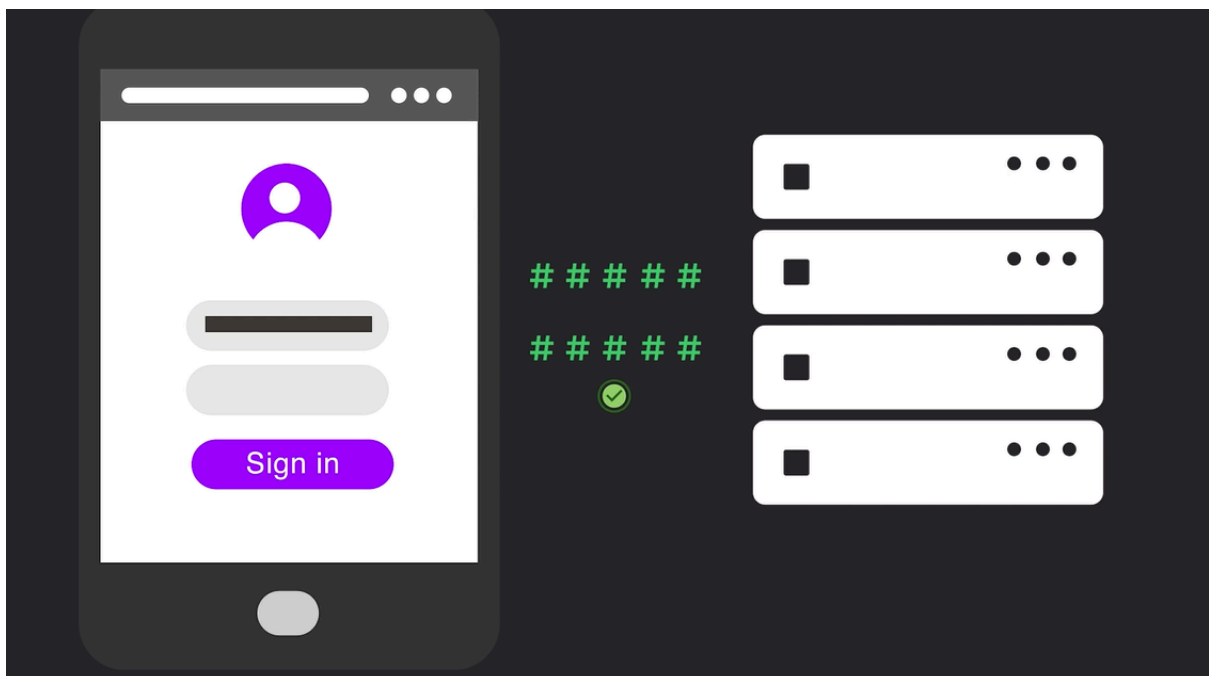
Nous enregistrons ensuite comme nous le ferions pour une route unique. Nous voulons enregistrer notre routeur pour toutes les demandes effectuées vers `/api/stuff` . Par

conséquent, tapez :

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

Préparez la base de données pour les informations d'authentification

Utilisation de Bcrypt pour crypter les logins des utilisateurs ⇒



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

```
npm install mongoose-unique-validator
```

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

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



```
const userSchema = mongoose.Schema({
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true }
});

userSchema.plugin(uniqueValidator);

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

Créez des utilisateurs

Configurez les routes d'authentification

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

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

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

};

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

};
```

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

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

```
const express = require('express');
const router = express.Router();

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

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

module.exports = router;
```

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

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

Puis enregistrez-le :

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

Créez des utilisateurs

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

```
npm install bcrypt
```

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

```
exports.signup = (req, res, next) => {  
  bcrypt.hash(req.body.password, 10)  
    .then(hash => {  
      const user = new User({  
        email: req.body.email,  
        password: hash  
      });  
      user.save()  
        .then(() => res.status(201).json({ message: 'Utilisateur créé !' }  
        )))  
      .catch(error => res.status(400).json({ error }));  
    })  
    .catch(error => res.status(500).json({ error }));  
};
```

Vérifiez les informations d'identification d'un utilisateur

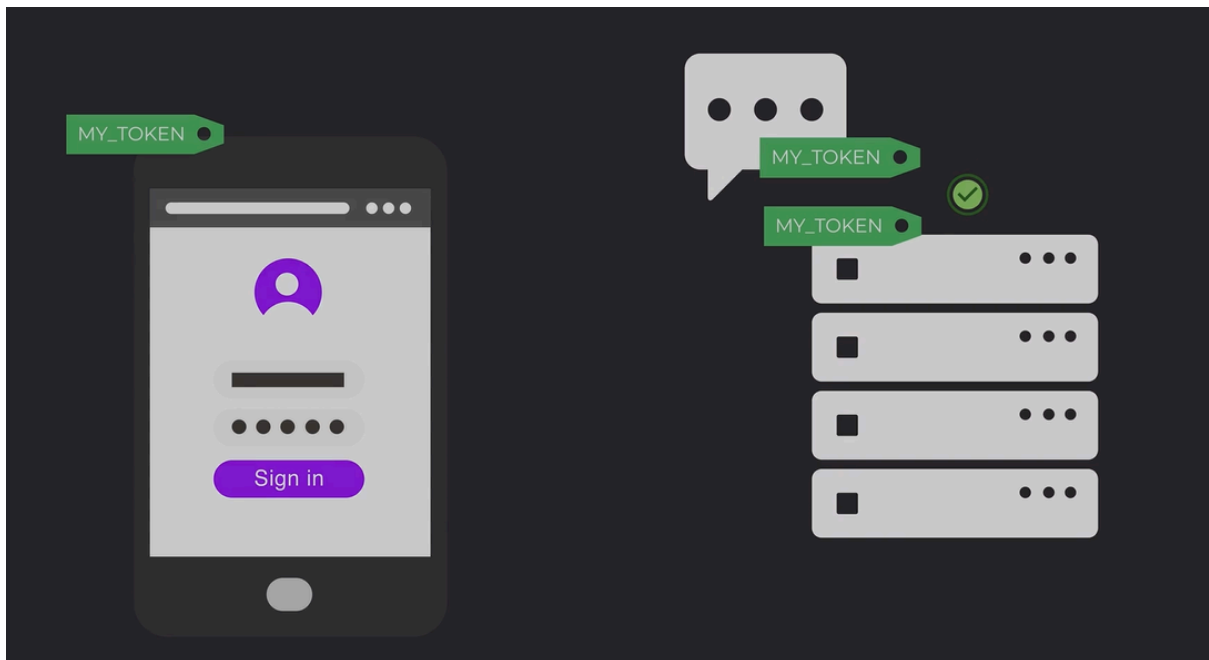
Implémentez la fonction login

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

```
exports.login = (req, res, next) => {
  User.findOne({ email: req.body.email })
    .then(user => {
      if (!user) {
        return res.status(401).json({ message: 'Paire login/mot de
        passe incorrecte' });
      }
      bcrypt.compare(req.body.password, user.password)
        .then(valid => {
          if (!valid) {
            return res.status(401).json({ message: 'Paire
            login/mot de passe incorrecte' });
          }
          res.status(200).json({
            userId: user._id,
            token: 'TOKEN'
          });
        })
        .catch(error => res.status(500).json({ error }));
    })
    .catch(error => res.status(500).json({ error }));
};
```

Créez des tokens d'authentification

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



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

```
npm install jsonwebtoken
```

Nous l'importerons ensuite dans notre contrôleur utilisateur :

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

Enfin, nous l'utiliserons dans notre fonction `login` :

```
exports.login = (req, res, next) => {
  User.findOne({ email: req.body.email })
    .then(user => {
      if (!user) {
        return res.status(401).json({ error: 'Utilisateur non trouvé' });
      }
      bcrypt.compare(req.body.password, user.password)
        .then(valid => {
          if (!valid) {
            return res.status(401).json({ error: 'Mot de'

```

```

    passe incorrect !' }));
  }
  res.status(200).json({
    userId: user._id,
    token: jwt.sign(
      { userId: user._id },
      'RANDOM_TOKEN_SECRET',
      { expiresIn: '24h' }
    )
  });
})
.catch(error => res.status(500).json({ error }));
})
.catch(error => res.status(500).json({ error }));
};

```

Configurez le middleware d'authentification

Implémentez le *middleware* d'authentification

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

```

const jwt = require('jsonwebtoken');

module.exports = (req, res, next) => {
  try {
    const token = req.headers.authorization.split(' ')[1];
    const decodedToken = jwt.verify(token, 'RANDOM_TOKEN_SECRET');
    const userId = decodedToken.userId;
    req.auth = {
      userId: userId
    };
    next();
  } catch(error) {
    res.status(401).json({ error });
  }
};

```

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

```
const express = require('express');
const router = express.Router();

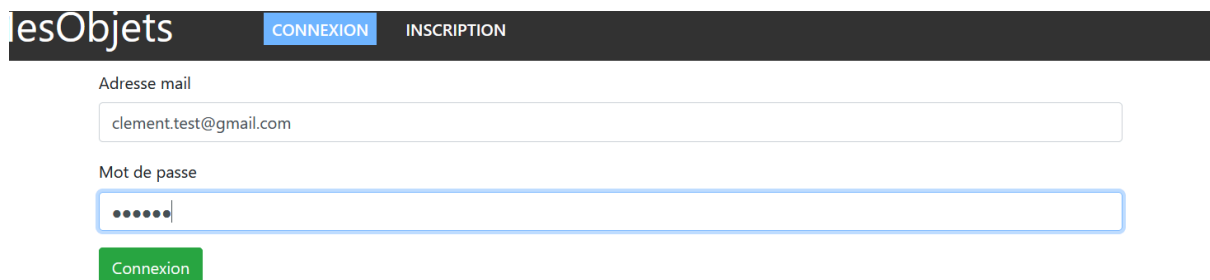
const auth = require('../middleware/auth');

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

router.get('/', auth, stuffCtrl.getAllStuff);
router.post('/', auth, stuffCtrl.createThing);
router.get('/:id', auth, stuffCtrl.getOneThing);
router.put('/:id', auth, stuffCtrl.modifyThing);
router.delete('/:id', auth, stuffCtrl.deleteThing);

module.exports = router;
```

Test de fonctionnalité :



The screenshot shows a web application interface for 'esObjets'. At the top, there is a dark header with the logo 'esObjets' on the left and two buttons, 'CONNEXION' and 'INSCRIPTION', on the right. Below the header, the login form is displayed. It has two input fields: 'Adresse mail' with the value 'clement.test@gmail.com' and 'Mot de passe' with masked characters '.....'. Below the password field is a green button labeled 'Connexion'.

nous sommes bien connecté :

Importer les marque-... Gmail Université SAE 1-03 - Google Dri... Portfolio - Google She... Accueil - Université Je... Padlet protégé par un ... (2) Home / Twitter Trainee Train Guard | 1... BUT R&T - IUT Réseau...

VendreMesObjets

OBJETS A VENDRE VENDRE UN OBJET DECONNEXION RETOUR A L'INDEX

LE MEILLEUR ENDROIT
POUR VENDRE VOS
OBJETS

 <p>Rolex €52,251.00</p>	 <p>apareil photo €2.00</p>
--	--

Acceptez les fichiers entrants avec multer

Nous devons d'abord installer `multer` dans notre projet :

```
npm install multer
```

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

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

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

const MIME_TYPES = {
  'image/jpg': 'jpg',
  'image/jpeg': 'jpg',
  'image/png': 'png'
};

const storage = multer.diskStorage({
  destination: (req, file, callback) => {
    callback(null, 'images');
  },
  filename: (req, file, callback) => {
    const name = file.originalname.split(' ').join('_');
    const extension = MIME_TYPES[file.mimetype];
    callback(null, name + Date.now() + '.' + extension);
  }
});

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


Modifiez les routes pour prendre en compte les fichiers

Modifiez la route POST

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

```
const express = require('express');
const router = express.Router();

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

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

router.get('/', auth, stuffCtrl.getAllThings);
router.post('/', auth, multer, stuffCtrl.createThing);
router.get('/:id', auth, stuffCtrl.getOneThing);
router.put('/:id', auth, stuffCtrl.modifyThing);
router.delete('/:id', auth, stuffCtrl.deleteThing);

module.exports = router;
```

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

```
exports.createThing = (req, res, next) => {
  const thingObject = JSON.parse(req.body.thing);
  delete thingObject._id;
  delete thingObject._userId;
  const thing = new Thing({
    ...thingObject,
    userId: req.auth.userId,
    imageUrl:
`${req.protocol}://${req.get('host')}/images/${req.file.filename}`
  });

  thing.save()
    .then(() => { res.status(201).json({message: 'Objet enregistré !'})})
    .catch(error => { res.status(400).json( { error })})
};
```

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')));
```