



Commencer avec Node.js, Express.js et MongoDB



Adama Samassa

Préface

Dans ce cours, nous vous emmenons dans un voyage pratique et pragmatique pour apprendre le développement Node.js, Express et MongoDB.

Vous commencerez à construire votre première application Node.js en quelques minutes.

Au final, vous aurez les compétences pour créer une application de blog et la déployer sur Internet.

Au cours de ce livre, nous aborderons :

- Chapitre 1 Introduction
- Chapitre 2 : Introduction à npm & Express
- Chapitre 3 : Commencer notre projet de blog
- Chapitre 4 : Moteurs de modèles
- Chapitre 5 : Introduction à MongoDB
- Chapitre 6 : Application de MongoDB à notre projet
- Chapitre 7 : Télécharger une image avec Express
- Chapitre 8 : Introduction à Express Middleware
- Chapitre 9 : Refactorisation en MVC
- Chapitre 10 : Enregistrement de l'utilisateur
- Chapitre 11 : Authentification des utilisateurs avec les sessions express
- Chapitre 12 : Affichage des erreurs de validation
- Chapitre 13 : Relier la collecte de messages à la collecte d'utilisateurs
- Chapitre 14 : Ajout d'un éditeur WYSIWYG
- Chapitre 15 : Utilisation de MongoDB Atlas
- Chapitre 16 : Déploiement d'applications Web sur Heroku

Le but de ce cours est de vous apprendre le développement de manière agréable sans vous submerger avec Node.js, Express et MongoDB

Nous nous concentrons uniquement sur l'essentiel et couvrir le matériel d'une manière pratique.

Chap. 1 : Introduction

Node.js est l'un des frameworks côté serveur les plus populaires. Beaucoup d'entreprises construisent leurs applications avec Node.js par exemple, Wal-Mart, LinkedIn, PayPal, YouTube, Yahoo!, Amazon.com, Netflix, eBay et Reddit.

Dans ce cours, nous allons découvrir Node.js avec Express et MongoDB et créer une application de blog à partir de zéro avec eux. Le processus vous aidera à passer du niveau débutant à celui où vous pourrez créer des applications efficacement en utilisant ces technologies.

Vous apprendrez une gamme de sujets tels que l'authentification des utilisateurs, la validation des données, le JavaScript asynchrone, le hachage de mot de passe, Express, MongoDB, des modèles moteurs, la maintenance des sessions utilisateur et plus encore.

L'application que nous allons créer

Nous allons créer une application de blog qui permettra aux utilisateurs d'écrire des entrées de blog une fois qu'ils se seront inscrits avec un compte.



Register a new account

User Name

user1

Password

••••

REGISTER

Title

	B	<u>U</u>		LORA ▾	A ▾			

Hello bootstrap 4

Choose file No file chosen

SEND

Une fois l'utilisateur enregistré, il peut accéder à la page d'accueil et se connecter avec ses identifiants.

La barre de navigation affichera dynamiquement différents éléments selon que l'utilisateur est connecté ou déconnecté.

Nous y parviendrons en utilisant le moteur de template EJS <https://ejs.co/>

Une fois connecté, la barre de navigation comportera un élément « Déconnexion ».

Il y aura également un élément « New Post » où l'utilisateur peut créer un nouveau billet de blog et télécharger un fichier image associé.

Quand il retourne à la page d'accueil, ses entrées de blog y seront publiées avec le nom d'utilisateur et la date postée.

Tout au long de cette application, nous allons apprendre beaucoup de concepts et solidifier nos connaissances Node.js, Express et MongoDB.

Qu'est-ce que Node.js ?

Avant de comprendre ce qu'est Node.js, nous devons comprendre comment Internet fonctionne.

Lorsqu'un utilisateur ouvre son navigateur et fait une demande à un site, il est connu comme client.

Le client fait une demande d'un certain site Web à un serveur qui, dès réception de la requête, répond avec le contenu du site demandé au client qui l'affiche ensuite dans son navigateur.

Par exemple, je fais une demande à amazon.com et les serveurs Amazon répondent avec le HTML pour amazon.com.

Il y a eu des langages de programmation côté serveur comme PHP, Ruby, Python, ASP, Java et autres qui nous aident à répondre aux demandes du navigateur correctement sur le serveur.

Traditionnellement, JavaScript est utilisé pour s'exécuter uniquement sur le navigateur pour fournir l'interactivité du site Web, par exemple des menus déroulants.

Mais en 2009, Node.js a récupéré le V8 JavaScript Engine de Google Chrome, hors du navigateur et lui a permis de s'exécuter sur des serveurs.

Ainsi, en plus des langages côté serveur existants, les développeurs peuvent désormais choisir JavaScript pour développer des applications côté serveur.

Quels avantages le choix de Node.js comme langage côté serveur apporte-t-il ?

- Premièrement, le moteur JavaScript V8 utilisé dans Google Chrome est rapide et peut exécuter des milliers d'instructions par seconde
- Deuxièmement, Node.js encourage la création d'un style de codage asynchrone pour un code plus rapide pour gérer la concurrence tout en évitant les problèmes du multithread.

- Troisièmement, en raison de sa popularité, JavaScript offre à Node.js un accès à de nombreuses bibliothèques utiles
- Et bien sûr, Node.js nous offre la possibilité de partager du code entre navigateur et serveur car ils utilisent tous les deux du code JavaScript.

Basé sur ceci, les développeurs ont créé la pile MEAN, un site Web entièrement JavaScript avec une pile d'applications constituée de MongoDB (une base de données contrôlée par JavaScript), Express, Angular (un framework JavaScript frontal et Node.js).

En fait, il y a également une pile MERN où au lieu d'utiliser Angular, on utilise React. Nous couvrirons MongoDB, Express et Node.js, c'est-à-dire la pile MEN.

Installation de Node.js

Pour installer Node.js, allez sur nodejs.org et téléchargez la version pour votre système d'exploitation.

Aller dans le terminal et vérifier l'installation :

```
node -v
```

Vous devriez aussi pouvoir vérifier pour npm :

```
npm -v
```

Création de notre premier serveur

Nous allons créer notre premier serveur pour mieux comprendre ce qu'est une demande d'un client et comment y répond le serveur. Dans un éditeur de code de votre choix, choisissez un emplacement de répertoire et créez-y un nouveau fichier appelé **index.js**.

Commencer par initialiser un projet avec :

```
npm init -y
```

Puis installez le module http dans les dépendances de votre projet :

```
npm i http --save
```

Remplissez le code ci-dessous pour créer notre premier serveur :

```
const http = require('http')
const server = http.createServer((req, res) =>{
  console.log(req.url)
  res.end('Hello Node.js')
})
server.listen(3000)
```


Explication du code

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

La fonction **require** dans Node.js nous aide à récupérer le package dans Node.js appelé **http**.

require existe aussi dans d'autres langages. **require** prend le nom d'un package comme argument et renvoie le package.

Nous avons besoin du package **http** et l'affectons à une variable appelée **http**.

Vous pouvez ensuite utiliser **http** comme vous le feriez pour n'importe quel objet. La convention générale est que lorsque vous avez besoin d'un module, vous le mettez dans une variable qui a le même nom que le module lui-même. Dans notre exemple, nous mettons le module **http** dans une variable du même nom **http**.

Vous pouvez bien sûr choisir de le mettre dans une variable avec un nom différent, rien ne l'empêche dans le code, mais cela évite la confusion.

http est l'un des packages intégrés fournis par Node.js pour effectuer des actions sur le serveur. D'autres packages intégrés essentiels que nous aborderons plus tard sont le package d'accès au système de fichiers (**fs**) et le package de fonction utilitaire (**util**).

```
const server = http.createServer(...)
```

Ensuite, nous créons et démarrons un serveur avec la méthode **createServer** à partir du paquet **http**. **createServer** prend en paramètre une fonction :

```
const server = http.createServer((req, res) =>{  
  console.log(req.url)  
  res.end('Hello Node.js')  
})
```

La fonction fournie à `createServer` est appelée fonction de rappel. Ce sera appelé lorsque la fonction `createServer` est terminée.

Quand cette fonction est appelée, elle recevra la demande (`req` - demande du navigateur) et la réponse (`res` - réponse à rendre au navigateur) objet dans la fonction.

Nous pouvons faire alors ce que nous voulons avec les deux objets dans le corps de la fonction.

Dans notre cas, nous donnerons simplement l'URL de la demande et après cela, enverrons le texte "Bonjour Node.js" dans le corps de la fonction.

```
server.listen(3000)
```

Enfin, avec `server.listen(3000)` nous démarrons notre serveur pour commencer à recevoir les requêtes.

C'est-à-dire que le serveur écoute les requêtes sur le port 3000.

Vous pouvez spécifier n'importe quel port que vous voulez, 3000 ou autre.

Si vous demandez ce qu'est un port ? Un port est une passerelle spécifique sur le serveur pour héberger une application particulière.

Par exemple, s'il y a plusieurs applications exécutées sur le même serveur, nous spécifions différents numéros de port pour différentes applications.

Exécution de *index.js*

Dans notre cas, pour toute demande faite au port 3000, nous répondons par 'Bonjour Node.js'.

Pour exécuter le fichier et lancer l'exécution du serveur, dans le Terminal, à partir du répertoire dans lequel se trouve le fichier et exécutez :

```
node index.js
```

Allez maintenant dans votre navigateur et entrez `http://localhost:3000/`.

`localhost` dans ce cas fait référence à notre ordinateur qui agit comme un serveur local.

Mais supposons que le serveur est hébergé sur un autre ordinateur ou site, vous pouvez imaginer que le lien serait `http://<computer ip>:3000/`.

Dans le dernier chapitre de ce livre, nous allons apprendre à déployer notre application sur un serveur externe Heroku pour créer notre application disponible sur Internet.

Dans votre navigateur, vous devriez voir le texte « Hello Node.js » affiché dans votre navigateur.

C'est parce que nous avons répondu à la demande avec le code

```
res.end(' Hello Node.js')
```

Si vous regardez votre terminal exécutant Node.js, vous pouvez voir ' / ' en cours de journalisation.

C'est parce que nous avons le code `console.log(req.url)` qui enregistre l'URL de la demande.

Si vous entrez `http://localhost:3000/noderocks`, '/noderocks' sera chargé dans le terminal. Ainsi, vous voyez que l'objet `req` contient des données de la requête dans le navigateur.

Nous avons maintenant un cycle de demande et de réponse réussi et j'espère que cet exemple vous a permis de mieux comprendre le fonctionnement d'une demande et d'une réponse entre un client et un serveur.

En savoir plus sur la demande et la réponse

Notre application ne répond actuellement qu'avec " Hello Node.js " quelle que soit l'url entré après localhost:3000. Pour avoir des réponses différentes en fonction de URL, ajoutez le code suivant en gras :

```
const http = require('http')
const server = http.createServer((req, res) =>{

  if(req.url === '/about')
    res.end('a propos')
  else if(req.url === '/contact')
    res.end('Page de Contact')
  else if(req.url === '/')
    res.end('Page Accueil')
  else {
    res.writeHead(404)
    res.end('page not found')
  }

})
server.listen(3000)
```

Pour exécuter le code nouvellement ajouté, nous devons arrêter et redémarrer le serveur avec `node index.js`

Explication du code

En utilisant une instruction if-else dans la fonction de rappel, nous vérifions la demande url et selon son chemin, nous répondons avec différents messages.

Si l'url contient « /a propos », nous servons la page à propos de. S'il contient ' /contact ', nous servons la page de contact et si c'est juste ' / ', nous servons la page d'accueil.

Si le chemin n'existe pas dans le if-else, nous utilisons par défaut la dernière clause else et répondons avec 'page not found' et aussi `writeHead(404)`.

`writeHead` écrit le code d'état de la requête. Normalement, un code d'état 200 indique que le serveur a répondu avec une réponse réussie.

Vous pouvez voir le code d'état de votre demande chaque fois que vous demandez un site au Navigateur Chrome en allant dans « Outils de développement sous « Vue », " Développeur " , " Outils de développement " puis "Network"

Si je demande une URL invalide comme « /contacts » (avec un « s » supplémentaire), elle retourne le code 404 indiquant 'Not Found'

Avec cela, nous avons un moyen de traiter différentes demandes d'un client et d'envoyer la réponse appropriée du serveur.

Répondre en HTML

Nous avons répondu aux demandes avec du texte statique. Dans un environnement réel, nous voulons répondre avec du HTML. Nous allons voir comment procéder dans cette section.

Dans le même dossier que `index.js`, créez un nouveau fichier appelé **index.html** avec le sous le code HTML simple : `<h1>Page d'accueil</h1>`

Faites de même pour :

about.html(`<h1>À propos de la page</h1>`),
contact.html(`<h1>Contact Page</h1>`) et
notfound.html(`<h1>Page introuvable</h1>`).

De retour dans `index.js`, ajoutez le code ci-dessous en gras :

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

const fs = require('fs')
const homePage = fs.readFileSync('index.html')
const aboutPage = fs.readFileSync('about.html')
const contactPage = fs.readFileSync('contact.html')
const notFoundPage = fs.readFileSync('notfound.html')

const server = http.createServer((req, res) =>{
  if(req.url === '/about')
    res.end(aboutPage)
  else if(req.url === '/contact')
    res.end(contactPage)
  else if(req.url === '/')
    res.end(homePage)
  else {
    res.writeHead(404)
    res.end(notFoundPage)
  }
})
server.listen(3000)
```

Explication du code

```
const fs = require('fs')
```

Nous importons un module file system 'fs' qui nous aide à interagir avec les fichiers sur notre serveur.

```
const homePage = fs.readFileSync('index.html')
const aboutPage = fs.readFileSync('about.html')
const contactPage = fs.readFileSync('contact.html')
const notFoundPage = fs.readFileSync('notfound.html')
```

La méthode `readFileSync` de `fs` lit le contenu de chaque fichier et le renvoie. Nous stockons le contenu de chaque page dans une variable.

```
const server = http.createServer((req, res) =>{
  if(req.url === '/about')
    res.end(aboutPage)
  else if(req.url === '/contact')
    res.end(contactPage)
  else if(req.url === '/')
    res.end(homePage)
  else {
    res.writeHead(404)
    res.end(notFoundPage)
  }
})
```

A la place de `res.end()` contenant un texte statique, `res.end` renvoie maintenant le contenu HTML de la variable.

Exécuter votre application

Redémarrez le serveur avec `node index.js` et nous aurons HTML présenté à la place. Et c'est ainsi que nous répondons aux requêtes avec HTML en utilisant le module de système de fichiers.

Notez que nous avons écrit une seule fonction JavaScript pour l'ensemble de notre application:

Cette fonction unique écoute les requêtes d'un navigateur Web, soit à partir d'un ordinateur, d'un mobile ou de tout autre client appelant notre API (Application Programming Interface).

Nous appelons cela **un gestionnaire de requêtes**. Lorsqu'une demande arrive, cette fonction examine la demande et décide de la manière d'y répondre.

Il faut deux arguments, un objet qui représente la demande (`req`) et un objet qui représente la réponse (`res`).

Chaque application Node.js est ainsi, une seule fonction du gestionnaire de requêtes peut répondre aux demandes.

Pour les petits sites, cela peut sembler facile, mais les choses peuvent rapidement devenir ingérables comme vous pouvez l'imaginer, par exemple un site comme Amazon.com qui inclut le rendu HTML réutilisable dynamique avec des modèles, du téléchargement d'images, etc. Nous explorons dans le chapitre suivant comment Express aide à résoudre ce problème et nous permet d'écrire plus facilement les applications Web avec Node.js.

En résumé

Nous allons créer une application de blog avec Node.js, Express et MongoDB.

Node.js nous permettra d'utiliser JavaScript comme langage de programmation côté serveur, ce qui nous donne des avantages comme une exécution rapide et la possibilité de partager du code entre le serveur et le client.

Nous avons compris comment une demande s'effectue et comment y répondre avec le cycle entre un client et un serveur en démarrant un simple exemple de serveur.

Nous avons traité les demandes et répondu de manière appropriée avec les deux manières : texte et HTML.

Chap. 2 Introduction à npm & Express

Installer des paquets avec npm

Dans notre application du chapitre un, nous avons importé des packages de Node.js lui-même, par exemple, http, fs.

Maintenant, la communauté Node.js compte de nombreux développeurs qui écrivent des modules/packages personnalisés que nous pouvons utiliser dans nos propres applications.

Ils sont hébergés sur un site appelé npmjs.com où vous pouvez aller pour voir ce qu'il est approprié d'utiliser dans votre propre code.

Nous utilisons npm (ou Node Package Manager) pour gérer les packages que nous téléchargeons. npm est un fonctionnaire assistant pour les projets Node qui accompagne Node lorsque nous l'avons installé.

Un package personnalisé très important que nous allons installer est Express.

Parce que les API vanilla Node.js peuvent être verbeuses, déroutantes et limitées en fonctionnalités, Express est un framework qui agit comme une couche légère au-dessus du serveur Web Node.js facilitant le développement d'applications Web Node.js.

Express est utilisé par de nombreuses entreprises. Nous verrons plus tard comment cela simplifie les API de Node.js, ajoute des fonctionnalités utiles, aide à organiser les fonctionnalités de notre application avec un middleware et du routage, ajoute des utilitaires utiles aux objets HTTP de Node.js et facilite le rendu des vues HTML dynamiques.

<http://expressjs.com/en/resources/frameworks.html>

Au cours de ce cours, nous allons explorer ces caractéristiques en profondeur.

Alors, allez sur npmjs.com et recherchez « express », il y aura des instructions sur comment installer et utiliser le package Express.

Comme mentionné dans le site sous « Installation », nous devons exécuter `npm install express` pour installer le package express.

Avant d'exécuter `npm install express`, npm nécessite que nous ayons un fichier `package.json` pour suivre tous les packages et leurs versions utilisés dans notre application.

Si vous exécutez `npm install express` maintenant sans le fichier, vous obtiendrez une série d'avertissements comme « aucun fichier ou répertoire de ce type, ouvrez... `package.json` ».

Pour générer le fichier `package.json`, exécutez « `npm init` » qui posera une série de questions sur notre projet (par exemple, nom du projet, auteur, version) pour nous créer le **`package.json`**.

Vous pouvez bien sûr créer manuellement le `package.json` de vous-même.

Mais `npm init` nous fait gagner un peu de temps lors de la création de fichiers `package.json`.

Maintenant, appuyez simplement sur Entrée pour toutes les questions et à la fin, `package.json` aura été généré pour nous.

Comme vous pouvez le voir, `package.json` contient des métadonnées sur notre projet Node comme le nom du projet, sa version et ses auteurs.

Ensuite, procédez à l'installation d'« express » avec `npm install express`.

Lorsque l'installation d'express est terminée, vous remarquerez qu'une nouvelle propriété dans les dépendances ont été ajoutées à `package.json`.

Les dépendances contiennent les packages de dépendances et leurs numéros de version.

Pour l'instant, nous avons la version 4.17.1 d'Express.

Chaque fois que nous installons un package, npm l'enregistre ici pour garder une trace des packages utilisés dans notre application

`npm install express` installe le package `express` dans notre application en récupérant la dernière version d'Express et en la plaçant dans un dossier appelé `node_modules`.

Si vous regardez votre dossier `app`, le dossier `node_modules` aura été créé pour vous. C'est là que les dépendances personnalisées sont enregistrées pour notre projet.

Si vous ouvrez et explorez `node_modules`, vous devriez pouvoir localiser le paquet `express`.

La raison pour laquelle nous voyons de nombreux autres packages dans `node_modules` même si nous n'avons installé qu'`express`, c'est parce que `express` dépend de ces autres packages et ils ont été installés lorsque `express` a été installé.

Ces autres packages sont des dépendances d'Express. Le fichier **`package-lock.json`** suit les versions de toutes les dépendances d'Express.

Présentation d'Express

Dans cette section, nous présentons Express et comment il rend l'application développement dans Node plus facile et plus rapide. Tout d'abord, nous importons express dans notre application en ajoutant le code ci-dessous à index.js :

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

Cela extrait le package du répertoire node_modules. Pour apprécier à quel point Express facilite le développement pour nous, nous allons utiliser Express pour répéter ce que nous avons déjà fait dans le premier chapitre.

Auparavant, nous avons créé un serveur et l'avons démarré avec :

```
const http = require('http')
const server = http.createServer((req, res) =>{
  ...
})
server.listen(3000)
```

C'est-à-dire que nous devons nous occuper d'importer http, fs, d'autres packages, ainsi que l'objet de requête et de réponse. Mais avec Express, nous obtenons la même chose avec :

```
const express = require('express') // récupère le module
express
const app = express() // créé une nouvelle fonction
express

app.listen(3000,()=>{
  console.log("App listening on port 3000")
})
```

Express prend en charge les objets **http**, **request** et **response** derrière la scène.

La fonction de ce rappel fournie dans le 2ème argument dans `app.listen()` est exécutée lorsque les serveurs commencent à écouter.

Exécuter votre application

Copiez le code Express ci-dessus dans `index.js`, exécutez `node index.js` dans la commande et voir le log « App écoute sur le port 3000 » affiché.

Nous en apprendrons plus sur les avantages de l'utilisation d'Express dans la section suivante.

Traitement des demandes avec Express

Express permet une plus grande flexibilité dans la réponse au navigateur.

Nous illustrons cela dans le code suivant en gras :

```
const express = require('express')
const app = express()
app.listen(3000,()=>{
  console.log("App à l'écoute sur le port 3000")
})

app.get('/',(req,res)=>{
  res.json({
    nom : 'Toto Gadget'
  })
})
```

Dans le code ci-dessus, vous pouvez voir qu'Express a rempli plus d'opérations pour que nous puissions mieux répondre aux demandes des navigateurs.

Par exemple, nous renvoyons une réponse JSON au navigateur avec `res.json`. Cela nous permet donc de construire des API avec Node.

Nous pouvons définir des routes spécifiques et les réponses du serveur donne lorsqu'un itinéraire est atteint, par exemple :

```
app.get('/about',(req,res)=>{  
  res.json({  
    nom : 'Jackie Jack'  
  })  
})
```

Ceci est également appelé Routage où nous mappons les demandes à des gestionnaires spécifiques selon leur URL. Auparavant, sans Express, nous avons dû répondre à des routes individuelles avec une instruction `if-else` étendue dans un gestionnaire de requête :

```
const server = http.createServer((req, res) =>{  
  if(req.url === '/about')  
    res.end(aboutPage)  
  else if(req.url === '/contact')  
    res.end(contactPage)  
  else if(req.url === '/')  
    res.end(page d'accueil)  
  autre {  
    res.writeHead(404)  
    res.end(notFoundPage)  
  }  
})
```

Avec Express, nous pouvons factoriser une fonction de gestionnaire de demandes en plusieurs gestionnaires de requêtes plus petits qui gèrent chacun une fonction spécifique.

Cela nous permet de construire notre application d'une manière plus modulaire et maintenable.

Nous pouvons également définir le type de réponse que le serveur sert. Par exemple, pour fournir des fichiers HTML avec express, nous ajoutons ce qui suit :

```
const path = require('path')
...
app.get('/', (req, res) => {
  res.sendFile(path.resolve(__dirname, 'index.html'))
})
```

Explication du code

```
const path = require('path')
```

Nous introduisons un nouveau module appelé **path**. **path** est un autre module intégré dans Node.

Il nous aide à obtenir le chemin spécifique de ce fichier, car si nous tapons

```
res.sendFile('index.html')
```

il y aura une erreur car un chemin absolu est nécessaire.

`path.resolve(__dirname, 'index.html')` aide en fournissant le chemin absolu complet qui change en fonction des différents systèmes d'exploitation.

Par exemple, sur Mac et Linux, nous nous référons à un répertoire utilisant `/directory`, mais sous Windows, nous nous y référons en utilisant `\directory`.

Le module **path** assure que les choses se passent bien, peu importe si le système d'exploitation est Windows, Mac ou Linux.


```
app.get('/', (req, res) => {  
  res.sendFile(path.resolve(__dirname, 'index.html'))  
})
```

Express ajoute également des choses comme la méthode `sendFile` qui, si nous devons implémenter sans Express, serait d'environ 45 lignes de fichier de code compliqué.

Vous voyez petit à petit comment Express facilite l'écriture de ces fonctions de gestionnaire de requêtes et en général simplifie le développement dans Node.

Actions Asynchrones avec les fonctions de rappel

Nous avons vu quelques exemples de code avec des fonctions de rappel.

Les fonctions de rappel sont un aspect important de Node.js qui aident à prendre en charge les tâches à faire de manière asynchrone. C'est-à-dire qu'au lieu d'attendre qu'une tâche soit terminée avant d'en exécuter une autre, par ex. en PHP :

Task 1 -> Task 2 -> Task 3 -> Task 4 -> Completion

Node.js permet d'effectuer des tâches en parallèle, où aucune tâche n'en bloque une autre.

Task 1 ->

Task 2 ->

Task 3 ->

Task 4 ->

Comment Node.js prend en charge le code asynchrone est avec des fonctions de rappel. Par exemple, les gestionnaires de requêtes ci-dessous n'ont pas à être exécutés de façon synchrone.

```
app.get('/', (req, res) => {  
  // requête database  
})  
app.get('/client', (req, res) => {  
  res.sendFile(path.resolve(__dirname, 'client.html'))  
})
```

C'est-à-dire que les requêtes pour « / » et « /about » arrivent ensemble, elles n'ont pas besoin qu'une demande s'achève avant une autre.

Les deux tâches peuvent commencer en même temps. Il se peut que la tâche qui interroge la base de données démarre et pendant que la base de données réfléchit, la deuxième demande 'about.html' peut être lancée.

Notre code ne fait pas deux choses à la fois, mais lorsqu'une tâche attend quelque chose, l'autre tâche peut s'exécuter.

Le code asynchrone comme les fonctions de rappel s'exécute donc beaucoup plus rapidement.

Diffuser d'autres fichiers HTML

Pour diffuser les fichiers HTML about et contact, ajoutez les éléments suivants :

```
app.get('/about', (req, res) => { // appelé lorsque la  
  demande de /about arrive  
  res.sendFile(path.resolve(__dirname, 'about.html'))  
})  
app.get('/contact', (req, res) => { // appelé lorsque la  
  demande à /contact arrive  
  res.sendFile(path.resolve(__dirname, 'contact.html'))  
})
```

Si vous redémarrez le serveur, notre application lancera le fichier about lorsqu'une demande de /about entre et le fichier de contact lorsqu'une demande à /contact arrive.

Notez ici que nous utilisons `app.get` pour gérer les requêtes HTTP GET qui est la méthode HTTP la plus courante. La requête GET comme son nom l'indique obtient des ressources comme une page d'accueil, une image etc.

Les méthodes GET ne changent pas l'état de notre app, il récupère simplement. D'autres méthodes comme POST et PUT le font et nous les couvriront plus tard.

Servir des fichiers statiques avec Express

Maintenant que nous servons des pages HTML à partir de notre serveur, comment pouvons-nous inclure des fichiers statiques tels que polices, images et CSS avec Express ?

Pour ce faire, dans `index.js` ajoutez la ligne en gras :

```
const express = require('express')
const app = express()
const path = require('path')
app.use(express.static('public'))

app.listen(3000,()=>{
  console.log("App à l'écoute sur le port 3000")
})
```

`app.use` est une fonction spéciale pour augmenter les fonctionnalités avec Express en ajoutant une fonction à la pile middleware de notre application.

Nous discuterons plus sur middleware dans un chapitre dédié plus loin.

express.static est un package expédié avec Express qui nous aide à servir des dossiers statiques.

Avec `express.static('public')`, nous spécifions que toute requête qui demande des éléments, doit les obtenir à partir du répertoire « public ».

Pour illustrer cela, créez d'abord un dossier appelé " public " dans notre dossier d'application.

A l'intérieur, créer les sous-dossiers « css » et « js » nous allons référencer ces deux dossiers dans notre fichier index.html.

Référencement d'*index.css*

Dans le dossier « css », créez un fichier appelé index.css avec le code suivant :

```
body {  
  background-color: coral;  
}
```

Pour référencer index.css dans index.html, ajoutez dans index.html la ligne suivante :

```
<link rel="stylesheet" href="css/index.css">
```

Notez que lorsque nous faisons référence à `href="css/index.css"`, Express fait référence à `/public/css/index.css`.

C'est-à-dire que l'URL du fichier statique servi est déterminée en combinant son URL avec " public " .

Référencement d'`index.js`

Pour créer un lien vers un fichier js externe, nous ajoutons de la même manière ce qui suit à `index.html` :

```
<script src="js/index.js"></script>
```

Cela suppose qu'il existe un sous-dossier de public appelé `js` contenant le fichier `index.js`.

Note importante : `index.js` auquel `index.html` fait référence dans ce cas est pour le côté client, pas `index` côté serveur.

Résumé

Nous avons appris à installer des packages personnalisés tiers à l'aide de `npm`.

`Express` est un noyau de package personnalisé qui facilite le traitement des demandes et envoie les réponses aux requêtes, facilitant ainsi le développement d'applications dans `Node` en général.

Nous avons généré le fichier `package.json` qui maintient nos métadonnées de projet en particulier ses dépendances de manière organisée.

Nous avons également appris à servir des fichiers statiques pour rendre les fichiers `CSS` et `JavaScript` disponibles pour le `HTML` fichiers clients.

Chap. 3 Mise en route du projet de blog

Télécharger le modèle

Comme nous avons déjà vu la conception de sites Web, du CSS ou de la façon de créer une interface utilisateur graphique, nous utiliserons un modèle de blog pour démarrer notre projet.

Nous utiliserons le thème « Clean Blog » de [startbootstrap.com](https://startbootstrap.com/themes/clean-blog/) (<https://startbootstrap.com/themes/clean-blog/>).

Téléchargez le fichier .zip qui contient le thème et extrayez-le dans un nouveau dossier pour ce projet.

Allez dans Terminal et cd dans le nouveau dossier. Exécutez `npm init` et dites « oui » à toutes les questions pour générer le package.json.

Ensuite dans le dossier, installez Express avec :

```
npm install express
```

Dans ce dossier, créer un fichier `index.js` qui servira de racine pour le projet. Mettre le code suivant :

```
const express = require('express')
const app = new express()
app.listen(4000, ()=>{
  console.log('App listening on port 4000')
})
```

ou mieux encore :

```
import express from 'express'
const app = new express()
app.listen(4000, ()=>{
  console.log('App listening on port 4000')
})
```

Puis démarrer le serveur `node index.js`

Redémarrage automatique du serveur avec nodemon

Pour l'instant, nous démarrons et arrêtons notre serveur à chaque fois que nous effectuons un changement de code dans index.js.

Maintenant, nous allons installer un package appelé nodemon qui détecte automatiquement les changements de code et redémarre le serveur afin que nous n'ayons pas pour l'arrêter et le redémarrer manuellement.

Installez nodemon avec la commande suivante :

```
npm install nodemon --save-dev
```

L'option --save permet d'avoir les dépendances répertoriées dans notre package.json donc que quelqu'un d'autre puisse installer les dépendances plus tard si nous lui donnons le projet - ils n'ont qu'à exécuter `npm install` sans arguments.

Vous pouvez aussi modifier manuellement package.json, puis exécutez `npm install` pour ajouter des dépendances.

Sans l'option --save, nous aurions le dossier node_modules mais il ne sera pas reflété dans notre package.json.

L'option -dev dit nous installons nodemon à des fins de développement seulement.

Nous ne voulons pas que nodemon fasse partie de la version de production de l'application. Lorsque nodemon est installé, dans package.json, vous pouvez le voir répertorié sous devDependencies. Cela indique que le package est destiné au développement uniquement et non inclus dans notre application lorsque nous la déployons.

Lancer npm

Nous allons démarrer notre application à partir d'un script npm avec npm start.

Allons dans package.json, allez dans « scripts » et apportez la modification suivante :

```
...  
"scripts": {  
  "start": "nodemon index.js"  
},...
```

Donc, au lieu d'exécuter notre application avec node index.js comme nous l'avons fait précédemment, nous l'exécutons maintenant avec : **npm start**.

npm start regarde dans notre fichier package.json, voyez que nous avons ajouté un script appelé start et exécutez la commande associée c'est-à-dire **nodemon index.js**. nodemon surveille nos fichiers pour les changements et redémarre automatiquement le serveur lorsqu'il y a des changements de code.

Nous pouvons bien sûr simplement exécuter nodemon index.js directement, mais parce que npm start est une convention où la plupart des serveurs Web Node peuvent être démarrés avec npm start. C'est-à-dire que nous n'avons pas besoin de connaître le nom réel du fichier si index.js ou app.js, nous démarrons simplement le serveur avec **npm start**.

dossier public pour servir des fichiers statiques

Ensuite, nous allons enregistrer un dossier public pour nos fichiers statiques comme ce que nous avons fait dans le chapitre précédent.

Dans index.js, ajoutez les éléments suivants :

```
const express = require('express')
const app= new express()
app.use(express.static('public'))
app.listen(4000, ()=>{
  console.log('App à l'écoute sur le port 4000')
})
```

Avec cela, Express s'attendra à ce que tous les actifs statiques soient dans le répertoire public.

Ainsi, créez un nouveau dossier appelé public dans le répertoire de l'application.

Déplacer les fichiers suivants à partir des fichiers modèles téléchargés en public :

index.html
about.html
contact.html
post.html

Ensuite, déplacez les dossiers de modèles téléchargés suivants vers le public :

css
img
js

Si vous vous demandez comment savoir quels fichiers et dossiers déplacer vers le dossier public, regardez dans **index.html** et voyez à quels actifs statiques il fait référence.

Voici les éléments pertinents de index.html ci-dessous :

Dans la balise head index.html, nous pouvons voir qu'il fait référence au dossier css.

Dans body, sous la barre de navigation, on voit qu'il fait référence à index.html, about.html, post.html et contact.html.

Ceux-ci doivent donc être en racine du dossier public.

Vous pouvez maintenant relancer le localhost en port 4000 et voir votre blog.

Et si vous cliquez sur les liens about, sample post et de contact, il devrait naviguer à ces pages aussi.

Création d'itinéraires de pages (Routing)

Actuellement, nous servons index.html, about.html, post.html et contact.html du dossier public, qui le traite comme n'importe quel autre fichier statique.

Nous devrions cependant servir ces fichiers en définissant des routes spécifiques avec **app.get** et réagir lorsque des itinéraires spécifiques sont pointés, comme nous l'avons fait auparavant.

Pour cela, créez un dossier **pages** dans le répertoire racine de l'application et déplacez tous les fichiers HTML du public là-bas.

Après avoir copié les fichiers, nous allons configurer les routes pour les différentes pages.

Pour configurer la route de la page d'accueil, nous enregistrons une route d'obtention pour servir la home HTML lorsque nous recevons une demande du navigateur.

Ajoutez ce qui suit dans index.js :

```
const express = require('express')
const path = require('path')
const app = new express()
...
app.get('/', (req, res) => {
  res.sendFile(path.resolve(__dirname, 'pages/index.html'))
})
```

Désormais, lorsqu'une requête est adressée à la route de la page d'accueil ' / ', index.html sera servi.

Essayez vous-même

Pouvez-vous essayer de créer les itinéraires pour about, sample post et contact dans votre code ?

Le code de routage doit être similaire à ce que nous avons créé pour notre homepage.

Dans index.js, créez la nouvelle route pour about, contact et sample:

```
app.get('/about', (req, res) => {
  res.sendFile(path.resolve(__dirname, 'pages/about.html'))
})
app.get('/contact', (req, res) => {
  res.sendFile(path.resolve(__dirname, 'pages/contact.html'))
})
app.get('/post', (req, res) => {
  res.sendFile(path.resolve(__dirname, 'pages/post.html'))
})
```

Notez que nous devons ajouter « pages/ » avant le fichier html car nous avons stocké les fichiers HTML dans le dossier pages.

Liens dans index.html

Maintenant, si vous exécutez votre application et essayez de naviguer vers about, contact et sample

Si vous postez des pages à partir de la barre de navigation, vous vous rendez compte qu'elles ne fonctionnent pas ou n'obtiennent pas de
Erreur « Cannot GET /contact.html ».

C'est parce que dans index.html, nous font actuellement référence à d'autres pages avec des liens href vers les fichiers html réels, c'est-à-dire about.html comme indiqué ci-dessous :

```
<li class="nav-item">
  <a class="nav-link" href="/">Home</a>
</li>
<li class="nav-item">
  <a class="nav-link" href="/about.html">About</a>
</li>
<li class="nav-item">
  <a class="nav-link" href="/post.html">Sample Post</a>
</li>
<li class="nav-item">
  <a class="nav-link" href="/contact.html">Contact</a>
</li>
```

Nous devons répéter le changement de lien ci-dessus pour la barre de navigation dans about.html, contact.html et post.html également, sinon les liens seront toujours rompus.

Vous vous rendrez vite compte qu'il s'agit d'un travail fastidieux et répétitif.

Et si nous devons ajouter un nouveau lien vers notre barre de navigation ? Cela signifie que nous devons répéter le même code pour la barre de navigation dans toutes nos pages. Vous vous rendez compte que c'est du travail en double ! Ce n'est pas évolutif et cela devient de plus en plus difficile à mesure que notre application grandit. Dans cette section suivante, nous allons résoudre ce problème en utilisant des moteurs de modèles (template engines).

Résumé

Nous avons commencé notre projet de blog avec un modèle de blog existant de startbootstrap.com et l'avons intégré dans un projet Node.js.

En utilisant le package nodemon, nous détectons automatiquement les changements de code dans notre projet et il redémarre le serveur.

Les itinéraires de page ont été créés pour servir la homepage, about et sample post.

Une barre de navigation a été créée pour contenir les liens vers ces itinéraires.

Chap. 4 Templating Engines

Les moteurs de modèles nous aident à rendre dynamiquement les pages HTML contrairement à avoir juste des pages statiques fixes qui dans le chapitre précédent nous ont posé un problème qui est dupliquer les destinations pour toutes les pages statiques.

Dans ce chapitre, nous allons refactoriser notre application pour utiliser un moteur de modèles qui nous permet d'abstraire notre application dans différents fichiers de mise en page afin que nous ne répétions pas encore le code commun servant toujours le même fichier HTML qu'avant.

Il existe de nombreux moteurs de modèles tels que Handlebars, Pug, etc.

Mais nous utiliserons 'EJS' (qui signifie Embedded JavaScript) car il est l'un des moteurs de modélisation les plus populaires et a été conçu par les personnes qui ont créé Express.

Tous ces différents moteurs de modélisation ont finalement le même but qui est de produire du HTML.

Comme indiqué sur son site, EJS est un langage de template simple qui nous permet de générer du HTML avec du JavaScript brut dans des balises de script simples, c'est-à-dire `<%= ... %>`.

Tout d'abord, installez EJS avec npm :

```
npm install ejs --save
```

Pour utiliser EJS dans notre application, dans index.js, ajoutez ce qui suit :

```
const express = require('express')
const path = require('path')
const app = new express()
const ejs = require('ejs')
app.set('view engine', 'ejs')
```

Avec `app.set('view engine','ejs')`, nous disons à Express d'utiliser EJS comme moteur de templating et que tout fichier se terminant par `.ejs` doit être rendu avec le package EJS.

Auparavant, nous répondions à une demande d'obtention avec les éléments suivants :

```
app.get('/', (req, res)=>{
  res.sendFile(path.resolve(__dirname, 'pages/index.html'))
})
```

Maintenant, avec EJS, nous faisons comme cela :

```
app.get('/', (req, res)=>{
  res.render('index');
})
```

Nous envoyons une vue à l'utilisateur en utilisant `res.render()`. Express ajoute la méthode de rendu à l'objet de réponse. `res.render('index')` cherchera dans un dossier 'views' pour le fichier `index.ejs`.

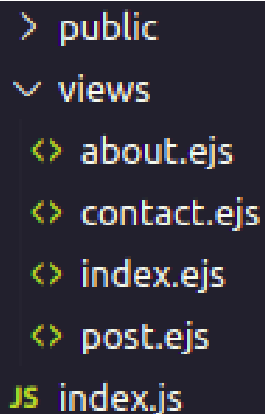
Ainsi, nous renommons notre dossier de pages actuel en `views`. Et dedans, renommez l'extension de fichier `index.html` en `index.ejs`.

Nous modifions également les gestionnaires de requêtes d'obtention avec `res.render` pour `about`, `contact` et `sample post` :

```
app.get('/about', (req, res) => {  
  //res.sendFile(path.resolve(__dirname, 'pages/about.html'))  
  res.render('about');  
})  
app.get('/contact', (req, res) => {  
  //res.sendFile(path.resolve(__dirname, 'pages/contact.html'))  
  res.render('contact');  
})  
app.get('/post', (req, res) => {  
  //res.sendFile(path.resolve(__dirname, 'pages/post.html'))  
  res.render('post');  
})
```

Finalement, modifiez les extensions de fichier `about.html`, `contact.html` et `post.html` en `about.ejs`, `contact.ejs` et `post.ejs`.

Nous devrions maintenant avoir un dossier `views` avec les fichiers EJS suivants :



```
> public  
v views  
  <> about.ejs  
  <> contact.ejs  
  <> index.ejs  
  <> post.ejs  
JS index.js
```


Dispositions : Layouts

Pour résoudre le problème du code répétitif (ex : barre de navigation, pied de page) apparaissant dans chaque page, nous utiliserons le concept de fichier de mise en page (**layout file**).

Un fichier de mise en page contient tout ce qui est commun aux pages, par exemple, la disposition de la barre de navigation, la disposition de l'en-tête, le footer, des scripts.... Chaque page inclura ensuite ces fichiers de mise en page dans en plus de leur propre contenu.

Il en résulte un texte très concis, lisible et fichier gérable.

Dans `index.ejs`, notez les éléments répétitifs, c'est-à-dire les éléments `<head>`, `<nav>`, `<footer>` et `<script>` qui apparaissent également dans les autres vues.

Notre but est d'extraire ces portions de code HTML commun dans leur propre fichiers de mise en page, c'est-à-dire `header.ejs`, `navbar.ejs`, `footer.ejs` et `scripts.ejs`.

Nous pourrions alors inclure les fichiers qui en ont besoin au lieu de répéter tout le code réduisant ainsi de beaucoup l'encombrement du code.

Tout d'abord, créez un sous-dossier dans **views** appelées **layouts** pour stocker ces fichiers.

Ensuite, extrayez le HTML `<head>` dans `header.ejs`, pareil pour `<nav>`, `<footer>` et `<scripts>`.

Après avoir extrait l'en-tête, la navigation, le pied de page et les scripts dans les différents fichiers de mise en page, **index.ejs** inclut désormais les différents fichiers de mise en page à la place du code extrait comme montré :

```
<!DOCTYPE html>
<html lang="en">
<%- include('layouts/header'); -%>

<body>
<%- include('layouts/navbar'); -%>
<!-- Page Header -->
<header class="masthead" style="background-image:
url('img/home-bg.jpg')">
<div class="overlay"></div>
<div class="container">
<div class="row">
<div class="col-lg-8 col-md-10 mx-auto">
<div class="site-heading">
<h1>Clean Blog</h1><span class="subheading">A Blog Theme by
Start Bootstrap</span>
</div>
</div>
</div>
</div>
</header>
<!-- Main Content -->
...
<hr>
<%- include('layouts/footer'); -%>
<%- include('layouts/scripts'); -%>
</body>
</html>
```

Explication du code

```
<%- include('layouts/header'); -%>
```

L'appel d'include reçoit un chemin relatif au modèle.

Ainsi, puisque le chemin de header.ejs est dans ./views/layouts/header.ejs et index.ejs est dans ./views/index.ejs, nous spécifions include('layouts/header').

Nous utilisons la balise de sortie brute <%- avec include pour permettre le rendu du HTML.

Pour voir la liste complète des balises disponibles dans EJS, reportez-vous à sa documentation dans <https://ejs.co/#docs> sous « Tags ».

Nous avons donc extrait la partie du code HTML commun dans sa propre mise en page fichier, c'est-à-dire en-tête, barre de navigation, pied de page, scripts.

Maintenant, nous pouvons simplement utiliser les déclaration include dans index.ejs, about.ejs, contact.ejs et post.ejs réduisant ainsi beaucoup de code répétitif.

Fichiers about, contact et post.ejs

Alors, procédez de la même manière pour les fichiers about.ejs, contact.ejs et post.ejs comme nous l'avons fait pour index.ejs.

Remarquez à quel point nos fichiers sont coupés maintenant ! Si tu lance l'application, tout devrait fonctionner comme avant.

Maintenant, si vous voulez changer le titre, allez simplement dans views/layouts/header.ejs et changez le:

```
<head>
<title>Super Clean Blog - Start Bootstrap Theme</title>
...
```

et cela se reflétera dans toutes les pages. Ou si je veux ajouter ou modifier un lien sur la barre de navigation, accédez simplement à **views/layouts/navbar.ejs** et modifiez-le.

Nous avons maintenant refactorisé notre application pour utiliser un moteur de modèles.

Les Templates Engines nous permettent d'abstraire notre application dans différents fichiers de mise en page afin de ne pas répéter le code commun, mais vous servez toujours le même fichier HTML qu'auparavant.

Nous verrons plus tard ce que d'autres moteurs de templates peuvent générer du contenu dynamique au fur et à mesure que nous construisons notre projet.

Résumé

Nous avons remanié notre code pour utiliser le moteur de modèles EJS pour rendre les pages HTML. Nous avons appelé `res.render` pour rendre dynamique les vues avec certaines variables.

Pour résoudre le problème de la répétition du code sur plusieurs pages par exemple la barre de navigation, nous avons utilisé les mises en page EJS.

Chap.5 Introduction à MongoDB

Nous utiliserons MongoDB comme base de données principale pour notre application. Vous pouvez, bien sûr, utiliser d'autres solutions pour conserver vos données d'application, par exemple, dans des fichiers, dans une base de données relationnelle SQL, ou dans un autre type de mécanisme de stockage.

Dans ce chapitre, nous couvrirons la base de données populaire MongoDB pour le stockage de base de données.

MongoDB est une base de données NoSQL. Avant de parler de ce qu'est une base de données NoSQL, commençons par parler des bases de données relationnelles afin que nous puissions observer un contraste significatif.

Si vous n'avez jamais entendu parler d'une base de données relationnelle auparavant, vous pouvez penser à des bases de données relationnelles comme des feuilles de calcul où les données sont structurées et chaque entrée est généralement une ligne dans un tableau.

Les bases de données relationnelles sont généralement contrôlées avec SQL ou Structured Query Language.

Les exemples de bases de données relationnelles populaires sont MySQL, SQL Server et PostgreSQL.

Les bases de données NoSQL sont souvent appelées bases de données non relationnelles, où NoSQL signifie tout ce qui n'est pas un SQL (voyez comment cela induit la popularité de SQL ?).

Il peut sembler que NoSQL est une protestation contre SQL, mais il fait en fait référence à une base de données non structurée comme un tableur, c'est-à-dire moins rigide que les bases de données SQL.

Alors, pourquoi utiliser Mongo ? Premièrement, il est populaire et cela signifie qu'il y a beaucoup d'aide en ligne.

Deuxièmement, il est mature depuis 2007 et utilisé par des entreprises comme eBay et Orange.

Architecture de MongoDB

Comme mentionné, l'architecture de MongoDB est une base de données NoSQL qui stocke des informations sous forme de collections et de documents.

MongoDB stocke une ou plusieurs collections. Une collection représente une entité unique dans notre app, par exemple dans une application de commerce électronique, nous avons besoin d'entités comme des catégories, utilisateurs, produits. Chacune de ces entités sera une collection unique dans notre base de données.

Une collection contient alors des documents. Un document est une instance de l'entité contenant les différents champs pertinents pour représenter le document.

Par exemple, un document produit contiendra des champs de nom, d'image et de prix. Chaque champ est une paire clé-valeur. Les documents ressemblent beaucoup à des objets JSON avec divers propriétés (bien qu'elles soient techniquement binaires JSON ou BSON).

Un exemple d'une arborescence de documents de collection est illustré ci-dessous :

Database

→ **Products** collection livres

→ Product document

```
{  
  price: 26,  
  title: "Learning Node",  
  description: "Top Notch Development book",  
  expiry date: 27-3-2020  
}
```

→ Product document

...

→ **Users** collection

```
→ User document
  {
    username: "123xyz",
    contact:
      {
        phone: "123-456-7890",
        email: "xyz@example.com"
      }
  }
→ User document
...
```

Installation de MongoDB

Il existe plusieurs façons d'installer MongoDB.

Si vous cherchez 'install mongodb', vous découvrirez différentes méthodes d'installation.

Vous choisissez votre préférée. Dans ce livre, je l'installe en utilisant brew pour installer l'édition communautaire MongoDB sur le système MacOS

(<https://docs.mongodb.com/manual/tutorial/install-mongodb-on-os-x/>).

Vous remarquerez que MongoDB nous recommande fréquemment d'utiliser son service cloud MongoDB Atlas.

Nous ne l'utiliserons pas pour l'instant, car nous voulons nous familiariser avec l'exécution de MongoDB sur notre propre serveur.

Mais plus tard, nous utiliserons MongoDB Atlas lorsque nous voulons déployer notre application pour la rendre disponible au monde.

Donc maintenant installez MongoDB selon votre OS.

Lancer MongoDB

Pour exécuter MongoDB (c'est-à-dire le processus mongod) au premier plan, exécutez la commande suivante :

```
mongod --config /usr/local/etc/mongod.conf
```

```
systemctl start mongod
```

Installation de MongoDB Compass

Ensuite, nous allons installer MongoDB Compass qui est un outil client fourni par l'équipe MongoDB elle-même pour nous aider à voir notre base de données visuellement.

Alors, soit google MongoDB Compass ou allez sur <https://www.mongodb.com/download-center/compass> pour le télécharger et l'installer. L'installation doit être directe.

Mongoose

Pour parler à MongoDB depuis Node, nous avons besoin d'une bibliothèque. La mongoose est un package Node.js officiellement pris en charge (<https://www.npmjs.com/package/mongoose>) qui nous aide à le faire. Pour l'installer, exécutez :

```
npm install mongoose
```

Se connecter à MongoDB depuis Node

```
const mongoose = require('mongoose');  
mongoose.connect('mongodb://localhost/my_database',  
{useNewUrlParser: true})
```


On définit une connexion avec `mongoose.connect` qui prend en paramètre un hôte et un nom de base de données.

Dans notre cas, parce que notre base de données s'exécute sur la machine locale, nous utilisons `localhost`.

Nous spécifions également le nom de la base de données que nous souhaitez vous connecter, dans notre cas `my_database`.

Lors de la connexion, MongoDB créera automatiquement cette base de données pour nous si elle n'existe pas.

Définir un modèle

Maintenant, dans le répertoire de votre application, créez un répertoire appelé **models**. Ce dossier contiendra des modèles qui sont des objets qui représentent des documents dans notre base de données.

Dans celui-ci, nous créons notre premier fichier modèle appelé **BlogPost.js** dans le dossier **models** avec le code suivant :

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema;
const BlogPostSchema = new Schema({
  title: String,
  body: String
});
```

Les modèles sont définis via l'interface **Schema**. N'oubliez pas qu'une collection représente une entité dans notre application. Par exemple, utilisateurs, produits, articles de blog.

Un schéma représente ce à quoi ressemble une collection. Cela signifie que chaque document de la collection aura les champs spécifiés dans le schéma.

Toujours dans BlogPost.js, continuez à ajouter le code suivant :

```
const BlogPost =  
mongoose.model('BlogPost', BlogPostSchema);  
module.exports = BlogPost
```

Nous accédons à la base de données via mongoose.model.

Le premier argument est le nom singulier de la collection à laquelle est destiné votre modèle.

Mongoose recherche automatiquement la version plurielle du nom de votre modèle. Dans notre cas, parce que nous utilisons BlogPost, Mongoose créera le modèle de notre collection BlogPosts, pas la collection BlogPost.

Enfin, nous exportons la variable BlogPost afin que lorsque d'autres fichiers nécessitent ces infos, ils peuvent "saisir" BlogPost. Notez que vous ne pouvez exporter qu'une seule variable.

Opérations CRUD avec les modèles Mongoose

Après avoir établi la connexion à MongoDB, nous illustrons ensuite CRUD (Create, Read, Update, Delete) via Mongoose.

Par souci de simplicité, nous allons illustrer CRUD dans un fichier de test séparé pour comprendre les concepts avant de les appliquer à notre projet.

Dans le répertoire de votre application, créez un fichier **test.js** avec le code suivant :

```
const mongoose = require('mongoose')
const BlogPost = require('./models/BlogPost')
mongoose.connect('mongodb://localhost/my_database',
  {useNewUrlParser: true, useUnifiedTopology: true}
);
BlogPost.create({
  title: 'Le super livre qui va nous servir de test',
  body: 'Il était une fois beaucoup de blabla'
}, (error, blogpost) =>{
  console.log(error, blogpost)
})
```

Explication du code

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

Nous importons le modèle BlogPost que nous venons de créer en spécifiant son chemin relatif.

BlogPost représente la collection BlogPosts dans la base de données.

```
mongoose.connect('mongodb://localhost/my_database',
  {useNewUrlParser: true});
```

Nous procédons ensuite à la connexion à la base de données. N'oubliez pas que si **my_database** n'existe pas, elle sera créée pour nous.

```
BlogPost.create({  
  title: 'Le super livre qui va nous servir de test',  
  body: 'Il était une fois beaucoup de blabla'  
}, (error, blogpost) =>{  
  console.log(error, blogpost)  
})
```

Nous créons ensuite un nouveau document BlogPost dans notre base de données avec une fonction **create** dans BlogPost.

Dans le premier argument, nous passons les données pour le document de billet de blog.

Dans le 2ème argument, nous passons une fonction de rappel qui est appelée lorsque **create** termine l'exécution. Mongoose nous donne n'importe quelle erreur dans l'argument d'erreur s'il y en a eu pendant l'opération de création.

Ça retourne aussi le message nouvellement créé dans l'argument blogpost.

Pour exécuter test.js, exécutez **node test.js** dans le terminal. Vous devriez voir null (erreur) et l'objet blogpost est enregistré comme ci-dessous :

```
null { _id: 60d5ec4a8b5fc192c180e917,  
  title: 'Le super livre qui va nous servir de test',  
  body: 'blabla',  
  __v: 0 }
```

Notez qu'il existe un champ supplémentaire `_id`. `_id` est un identifiant unique fourni par MongoDB pour chaque document.

Visualiser les données dans MongoDB

Pour voir les données visuellement dans MongoDB, ouvrez MongoDB Compass et insérer l'adresse de notre database :

```
mongodb://localhost/my_database
```

En dessous, vous devriez voir les articles de blog de la collection. Et si vous cliquez sur les articles de blog, vous pourrez voir le document de l'article de blog que nous venons de créer. Essayez d'insérer plus d'articles de blog.

Lecture des données de MongoDB à l'aide de Mongoose

Pour sélectionner tous les documents de la collection BlogPosts : transmettez un document vide en paramètre de filtre de requête au premier argument de la méthode find.

```
BlogPost.find({}, (error, blogspot) =>{  
  console.log(error, blogspot)  
})
```

Le paramètre de filtre de requête détermine les critères de sélection. Ainsi, pour trouver tous les documents dans la collection BlogPosts avec un titre particulier, par ex. 'Le super livre qui va nous servir de test', nous faisons :

```
BlogPost.find({  
  title: 'Le super livre qui va nous servir de test'  
}, (error, blogspot) =>{  
  console.log(error, blogspot)  
})
```

Ou, pour trouver tous les documents dans la collection BlogPosts avec "Le" dans le titre, nous faisons :

```
BlogPost.find({
  title:/Le/}, (error, blogspot) =>{
  console.log(error,blogspot)
})
```

C'est-à-dire que nous plaçons l'opérateur générique avant et après Le. Pour ceux qui sont familiers avec SQL, ' / ' agit comme ' % ' .

Pour obtenir des documents de base de données uniques, c'est-à-dire pour récupérer des documents uniques avec un identifiant unique `_id`, nous utilisons la méthode `findById` :

```
var id = "5cb436980b33147489eadfbb";
BlogPost.findById(id, (error, blogspot) =>{
  console.log(error,blogspot)
})
```

Il existe de nombreuses autres conditions de contraintes de recherche que vous pouvez appliquer pour rechercher.

Reportez-vous à <https://docs.mongodb.com/manual/tutorial/query-documents/> pour plus d'informations.

Mise à jour des enregistrements (UPDATE)

Pour mettre à jour un enregistrement, nous utilisons **findByIdAndUpdate** où nous fournissons l'identifiant comme premier argument et les champs/valeurs à mettre à jour dans le deuxième argument.

```
var id = "5cb436980b33147489eadfbb";
BlogPost.findByIdAndUpdate(id,{
  title:'Updated title'
}, (error, blogspot) =>{
  console.log(error,blogspot)
})
```

Effacer un enregistrement (DELETE)

Pour supprimer un enregistrement, nous utilisons le **findByIdAndDelete** où nous fournissons l'identifiant comme premier argument.

```
var id = "5cb436980b33147489eadfbb";
BlogPost.findByIdAndDelete(id, (error, blogspot) =>{
  console.log(error,blogspot)
})
```

Résumé

Nous avons découvert MongoDB, une base de données NoSQL qui stocke les données sous forme de collections et de documents.

En utilisant Mongoose, nous avons connecté notre application Node avec la base de données Mongo.

Nous avons défini des modèles en utilisant l'interface **Schema** pour représenter les collections dans notre base de données.

Nous avons illustré des opérations CRUD via Mongoose.

Enfin, nous avons utilisé MongoDB Compass pour voir les données stockées visuellement dans MongoDB.

Chap.6 APPLICATION DE MONGODB sur NOTRE PROJET

Après avoir introduit MongoDB, nous utilisons maintenant la base de données MongoDB pour construire notre application.

Tout d'abord, nous allons implémenter la fonctionnalité permettant aux utilisateurs de créer un nouveau message.

Pour cela, nous aurons besoin d'une vue pour que les utilisateurs puissent saisir les données.

Dans le dossier views, créez un nouveau fichier **create.ejs**. Comme point de départ, copiez le code de **contact.ejs** dans create.ejs.

Autrement dit, create.ejs comme les autres vues comprendra l'en-tête, la barre de navigation, le pied de page et les scripts.

Mais changez le texte `<h1>` à « Créer un nouveau message » :
Ensuite, enregistrez l'itinéraire pour « Créer un nouveau message » en ajoutant ce qui suit à **index.js** :

```
app.get('/posts/new', (req, res) => {  
  res.render('create')  
})
```

Ensuite, ajoutez « New Post » dans la barre de navigation. Dans `views/layouts/navbar.ejs`, ajoutez ce qui suit:

```
<li class="nav-item">
  <a class="nav-link" href="/posts/new">New Post</a>
</li>
```

Maintenant, si vous exécutez votre application et accédez à votre vue « New Post », la page devrait apparaître mais il semble qu'il ne puisse pas référencer les fichiers statiques comme css, images, js.

C'est parce que notre page est un itinéraire à deux niveaux et ne peut pas référencer les fichiers statiques requis par ex. dans `header.ejs`, car le href est :

Il suffit pour cela de rajouter un / à `css/styles.css` et à `js/scripts.js` dans `header.js` et `scripts.js`.

Avec cela, les routes à deux niveaux (et autres niveaux) peuvent avoir des références absolues à nos éléments statiques.

Maintenant, si vous exécutez votre application et accédez à votre vue « New Post », la page devrait avoir son style car elle est capable de référencer les fichiers statiques comme css, images et js.

Dans `create.ejs`, il existe déjà un formulaire de contact.ejs. A la place de **name**, nous le changeons en titre (titre du blog).

Parce que nous n'avons pas besoin des champs e-mail et de numéro de téléphone, supprimez-le.

Et renommez « Message » en corps (blog corps du contenu).

Par souci de simplicité, nous avons également supprimé certaines validations de formulaires comme `required` `data-validation-required-message="Veuillez entrer un message."` et des messages d'alerte (`<p class="help-block text-danger"></p>`) comme nous voulons nous concentrer sur la bonne forme de notre formulaire.

Nous reviendrons plus tard à la validation du formulaire et présentation des messages d'alerte. Le formulaire de création doit ressembler quelque chose comme :

```
<div class="container px-4 px-lg-5">
  <div class="row">
    <div class="col-lg-8 col-md-10 mx-auto">
      <form action="/posts/store" method="POST">
        <div class="control-group">
          <div class="form-group
floating-label-form-group controls">
            <label>Title</label>
            <input type="text"
class="form-control" placeholder="Title" id="title"
name="title">
          </div>
        </div>
        <div class="control-group">
          <div class="form-group
floating-label-form-group controls">
            <label>Description</label>
            <textarea rows="5"
class="form-control" id="body" name="body"></textarea>
          </div>
        </div>
        <br>
        <div class="form-group">
          <button type="submit" class="btn
btn-primary" id="sendMessageButton">Send</button>
        </div>
      </form>
    </div>
  </div>
</div>
```

Remarquez que vous avez besoin de l'attribut **name** pour chaque champ par ex. et également de ne PAS mettre les guillemets dans le type :

```
<input type=text class="form-control" placeholder="Title"
id="title" name="title" >
```

Et aussi :

```
<form action="/posts/store" method="POST">
```

Cela signifie que lorsque le formulaire est soumis, le navigateur fera une demande « POST » au point de terminaison /posts/store. Une requête « POST » est nécessaire pour passer un formulaire de données du navigateur à notre application Node.js pour créer l'enregistrement dans la base de données.

Dans **index.js**, ajoutez la fonction suivante pour gérer la requête POST :

```
app.post('/posts/store', (req, res) => {
  console.log(req.body)
  res.redirect('/')
})
```

Dans la fonction, nous obtenons les données du formulaire du navigateur via l'attribut **body**.

Mais pour permettre cela, nous devons d'abord installer un middleware qui analyse le **body** appelé **body-parser**. **body-parser** analyse les corps des requêtes entrantes dans un middleware et rend les données du formulaire disponibles sous la propriété req.body.

Ici, nous traitons une requête POST qui est généralement utilisée pour demander en plus l'état du serveur contrairement à GET où l'on obtient simplement des ressources.

Un utilisateur POSTE avec une entrée de blog, une photo, ou en s'inscrivant à un compte, en achetant un item etc.

POST est utilisé pour créer des enregistrements sur des serveurs.

Pour modifier un enregistrement existant, nous utilisons la requête PUT.

Alors, procédez à l'installation du **body-parser** en utilisant :

```
npm install body-parser
```

Ensuite ajouter ce code à **index.js** :

```
const bodyParser = require('body-parser')
app.use(bodyParser.json())
app.use(bodyParser.urlencoded({extended:true}))
```

Une autre possibilité depuis la version 4.17+ d'Express :

```
app.use(express.json());
app.use(express.urlencoded({
  extended: true
}));
```

Lancer l'application

Si vous exécutez votre application maintenant, allez dans « New Post », remplissez le formulaire et appuyez sur submit.

Vous serez redirigé vers la page d'accueil avec la méthode de redirection de la réponse (Express ajoute la méthode de redirection à l'objet de réponse pour nous - en utilisant uniquement Node, la redirection nécessiterait beaucoup plus de code).

Dans le terminal, vous verrez vos valeurs saisies dans un objet, par exemple :

```
{ title: 'title1', body: 'body1' }
```

C'est ainsi que l'on peut récupérer des données saisies dans un formulaire.

Avec `bodyParser`, un nouvel objet `body` contenant les données analysées est renseigné sur la demande objet.

Vous pouvez accéder à des propriétés individuelles dans l'objet corps, c'est-à-dire `req.body.title`, `req.body.body`.

Enregistrement des articles dans la base de données

Ayant les données du formulaire du navigateur dans `req.body`, nous l'utilisons maintenant avec le modèle `BlogPost` pour créer un nouveau document dans la collection `BlogPosts`, ou autrement dit, pour avoir nos données stockées dans la base de données.

Ajoutez ce qui suit dans **`index.js`** :

```
const BlogPost = require('./models/BlogPost.js')
...
app.post('/posts/store', (req, res) => {
  BlogPost.create(req.body, (error, blogpost) => {
    res.redirect('/')
  })
})
```

Nous importons le modèle `BlogPost`, appelons sa méthode `create`, fournissons `req.body` comme premier argument et une fonction de rappel comme 2ème argument qui est appelée lorsque la création est terminée.

Si vous exécutez votre application, remplissez les données dans le formulaire et appuyez sur `Submit`.

Vous pouvez voir vos données d'articles de blog nouvellement créées dans la collection `BlogPosts`.

Vous remarquerez peut-être que notre code pour créer un article de blog devient légèrement plus complexe dans la mesure où nous avons une fonction de rappel dans une fonction de rappel.

Lorsque notre application grandit et que nous effectuons davantage d'appels de méthode asynchrones, il peut arriver que nous ayons plus de couches de rappel.

C'est ce que nous appelons "l'enfer des rappels", (callback hell) où le code de rappel imbriqué est difficile à comprendre.

```
app.post('/posts/store', (req, res) => {  
  BlogPost.create(req.body, (error, blogpost) => {  
    res.redirect('/')  
  })  
})
```

Pour résoudre ce problème, nous pouvons également utiliser une fonctionnalité dans ES8 appelée **async** et **await** pour l'appel de méthode asynchrone comme indiqué dans le code suivant :

```
app.post('/posts/store', async (req, res) => {  
  await BlogPost.create(req.body)  
  res.redirect('/')  
})
```

Avec **async**, nous spécifions que la méthode suivante est un appel asynchrone.

Et en utilisant **await** pour **BlogPost.create**, nous disons que nous attendrons l'achèvement de la ligne en cours avant que la ligne ci-dessous puisse être exécutée.

Cette méthode nous permet ainsi d'avoir un code plus lisible.

Affichage d'une liste de billets de blog

Pour afficher la liste des articles de blog, nous utilisons la méthode `find` du modèle `BlogPost`

(nous avons déjà vu la méthode **`find`** dans la section « Introduction à MongoDB ») pour trouver tous les enregistrements de la base de données.

Nous le faisons chaque fois que la page d'accueil est demandée. Ainsi, dans `app.get` pour la page d'accueil, c'est-à-dire `'/'`, nous faisons :

```
app.get('/', async (req, res) => {
  const blogposts = await BlogPost.find({})
  res.render('index', {
    blogposts: blogposts
  });
})
```

Après avoir récupéré tous les articles du blog et les avoir affectés à la variable `articles` de blog, nous transmettons les données des articles de blog au navigateur client en le fournissant comme 2^{ème} argument à `render`.

Et chaque fois que le nom de la clé et nom de la valeur sont les mêmes (par exemple `blogposts : blogposts`), nous pouvons le raccourcir plus simplement:

```
app.get('/', async (req, res) => {
  const blogposts = await BlogPost.find({})
  res.render('index', {
    blogposts
  });
})
```

Avec cela, la vue `index.ejs` a maintenant accès à la variable `blogposts`.

Parce que nous mettons en console les articles de blog avec `console.log(blogposts)`, nous pouvons voir que `blogposts` est un tableau d'objets `BlogPost`.

Données dynamiques avec Templating Engines

Maintenant que nous avons les données des articles de blog renvoyées dans un tableau, nous allons utiliser notre Moteur de template EJS pour afficher dynamiquement les articles de blog dans la homepage.

Dans `index.ejs`, vous voyez actuellement des balises

`<div class="post-preview">` répétées chacune représentant un article de blog.

Ainsi, nous allons parcourir le tableau `blogposts` dans une boucle `for` et restituer une balise `<div class="post-preview">` pour chaque article de blog avec ce qui suit en gras :

```
<div class="container">
  <div class="row">
    <div class="col-lg-8 col-md-10 mx-auto">
      <% for (var i = 0; i < blogposts.length; i++) { %>
----><div class="post-preview">
      <a href="post.html">
        <h2 class="post-title">
          <%= blogposts[i].title %>
        </h2>
        <h3 class="post-subtitle">
          <%= blogposts[i].body %>
        </h3>
      </a>
      <p class="post-meta">Posted by
        <a href="#">Start Bootstrap</a>
        on September 24, 2019</p>
      </div>
      <hr>
-----><% } %>
    </div>
```

Explication du code

```
<% for (var i = 0; i < blogposts.length; i++) { %>
```

Avec le tableau `blogposts` fourni, nous utilisons une boucle `for` pour parcourir ses éléments.

```
<% for (var i = 0; i < blogposts.length; i++) { %>
<div class="post-preview">
  <a href="post.html">
    <h2 class="post-title">
      <%= blogposts[i].title %>
    </h2>
    <h3 class="post-subtitle">
      <%= blogposts[i].body %>
    </h3>
  </a>
  <p class="post-meta">Posted by
  <a href="#">Start Bootstrap</a>
  on September 24, 2019</p>
</div>
<hr>
<% } %>
```

Nous remplissons ensuite :

- titre de l'article de blog sous la balise `<h2 class="post-title">`,
- corps de l'article de blog sous `<h3 class="post-subtitle">`

Si vous exécutez votre application maintenant, vous verrez les articles de blog affichés dynamiquement

Il y a bien sûr d'autres champs que nous aurions pu remplir pour chaque blog poste, par exemple :

- itinéraire individuel des articles de blog dans ``,
- nom de l'auteur de l'article de blog dans `<p class="post-meta">Publié par ...`
- et la date de publication du blog le `</p>`

Nous travaillerons pour renseigner le nom et la date de l'auteur de l'article de blog dans les sections suivantes.

Pour l'itinéraire de l'article de blog, remplissez-le comme suit :

```
<div class="post-preview">
  <a href="/post/<%= blogposts[i]._id%>">
    ...
  </a>
  ...
</div>
```

Dans la section suivante, nous travaillerons à générer des pages uniques pour chaque article de blog.

Exercice

Maintenant que nous avons implémenté la liste de tous les articles de blog dans la page d'accueil, vous allez essayer comme exercice d'ajouter une barre de recherche et répertorier uniquement les articles de blog qui correspondent les termes de recherche :

Au début, nous ne fournirons aucun critère de recherche à `BlogPost.find({}...)` pour obtenir tous les postes.

```
BlogPost.find({}, (error, blogspot) =>{
  console.log(error, blogspot)
})
```

Maintenant, pouvez-vous fournir des paramètres de filtre de requête pour déterminer les critères de recherche ?

Par exemple. pour trouver tous les documents dans la collection `BlogPosts` avec un titre particulier ?

```
BlogPost.find({
  title: 'Le titre à rechercher'
}, (error, blogspot) =>{
  console.log(error, blogspot)
})
```

Ou, pour trouver tous les documents dans la collection BlogPosts avec "Le " dans le titre ?

```
BlogPost.find({
  title: /Le/}, (error, blogspot) =>{
  console.log(error, blogspot)
})
```

Page d'article de blog unique

Nous voulons afficher les détails de chaque article de blog dans leur propre URL spécifique. Premièrement, nous devons modifier la définition de l'itinéraire pour un seul article de blog. Actuellement en index.js, app.get('/post ... est :

```
app.get('/post', (req, res) =>{
  res.render('post')
})
```

Changez le en :

```
app.get('/post/:id', async (req, res) =>{
  const blogpost = await BlogPost.findById(req.params.id)
  res.render('post', {
    blogpost
  })
})
```

Explication du code

```
app.get('/post/:id'...
```

Tout d'abord, nous ajoutons un paramètre à la route pour un seul message. **id** représente un caractère générique qui accepte n'importe quelle valeur de chaîne, par ex.

http://localhost:4000/post/5cb836f610d8d629530fcf82.

Dans notre cas, nous avons une id donnée avec blogpost `_id` c'est-à-dire `<a href="/post/<%= blogposts[i]._id %>">`.

Les paramètres après `/post/` peuvent être récupérés avec `req.params`. Par exemple, si nous lançons ce code :

```
app.get('/post/:id', async (req, res) => {  
  console.log(req.params)  
})
```

Nous obtenons l'objet `params` en cours d'impression :

```
{ id : '5cb836f610d8d629530fcf82' }
```

C'est-à-dire qu'il affiche la clé et la valeur du paramètre dans l'URL.

Avec le param `id`, nous l'utilisons pour appeler `BlogPost.findById` qui récupère le post avec cet identifiant et transmet la variable `blogpost` à `post.ejs`.

```
app.get('/post/:id', async (req, res) => {  
  const blogpost = await BlogPost.findById(req.params.id)  
  res.render('post', {  
    blogpost  
  })  
})
```

post.ejs

Pour afficher dynamiquement les données uniques de chaque publication dans post.ejs, faites les modifications suivantes en gras :

```
<div class="col-lg-8 col-md-10 mx-auto">
  <div class="post-heading">
    <h1><%= blogpost.title %></h1>
    <h2 class="subheading"><%= blogpost.body %></h2>
    <span class="meta">Posted by
      <a href="#">Start Bootstrap</a>
      on August 24, 2019</span>
```

```
<!-- Post Content -->
<article>
  <div class="container">
    <div class="row">
      <div class="col-lg-8 col-md-10 mx-auto">
        <%= blogpost.body %>
      </div>
    </div>
  </div>
</article>
<hr>
<%- include('layouts/footer'); -%>
<%- include('layouts/scripts'); -%>
```

Explication du code

Pour afficher dynamiquement les données uniques de chaque article, nous modifions le titre dans la balise `<h1>` avec :

```
<h1><%= blogpost.title %></h1> ,
```

remplacez le sous-titre dans `<h2>` par

```
<h2 class="subheading"><%= blogpost.body %></h2>
```

et le corps de l'article par

```
<%= blogpost.body %>.
```

Lancer l'application

Maintenant, lorsque vous exécutez votre application et cliquez sur un message spécifique de la page d'accueil, la page d'un post sera remplie avec les données de cet article de blog spécifique.

Si vous cliquez sur un autre message, le titre et le corps changeront dynamiquement pour cela les données de l'article de blog.

Ajout de champs au Schema

Étant donné que nous n'avons actuellement que les champs titre et corps de notre article de blog, nous avons besoin maintenant d'un champ nom **User** et un champ **datePosted** pour savoir qui et quand pour la création du post.

Pour ajouter les champs, ajoutez les codes en gras ci-dessous :

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema;
const BlogPostSchema = new Schema({
  title: String,
  body: String,
  username: String,
  datePosted:{
    type: Date,
    default: new Date()
  }
});
```

Avec les champs ajoutés, apportez des modifications dans **index.ejs** et **post.ejs** :

index.ejs

```
<p class="post-meta">Posted by
  <a href="#"><%= blogposts[i].username %></a>
  on <%= blogposts[i].datePosted.toDateString() %></p>
</div>
<hr>
<% } %>
<!-- Pager -->
```

post.ejs

```
<h1><%= blogpost.title %></h1>
<h2 class="subheading"><%= blogpost.body %></h2>
<span class="meta">Posted by
-->    <a href="#"><%= blogpost.username %></a>
-->    on <%= blogpost.datePosted.toDateString() %></span>
</div>
```


`</div>`

Supprimez maintenant les enregistrements dans la base de données et réinsérez les enregistrements afin que notre article de blog contienne les champs supplémentaires.

Cette fois, `datePosted` sera automatiquement fournie. Plus tard, nous remplirons le champ du nom d'utilisateur lorsque nous implémenterons la fonction de connexion.

Résumé

Nous avons utilisé MongoDB pour créer notre application de blog. Nous avons mis en place un formulaire pour créer un article de blog et utilisé le package `body-parser` (ou celui implémenter par `express`) pour récupérer les données du champ de formulaire.

Le modèle `BlogPost` a été utilisé pour stocker les données dans la base de données. Nous avons affiché la liste des articles de blog dans la page d'accueil avec le moteur de template EJS et avons affiché également les détails de chaque article de blog dans leurs propres pages individuelles.

Chap. 7 Uploader une image avec Express

Dans ce chapitre, nous allons explorer comment télécharger une image pour un article de blog.

Nous installerons un package `express-fileupload` pour aider à télécharger des fichiers dans Express

(<https://www.npmjs.com/package/express-fileupload>).

Comme indiqué dans sa documentation, exécutez la commande suivante pour installer `express-fileupload` :

```
npm install --save express-fileupload
```

Ensuite, ajoutez un champ de téléchargement de fichier dans notre formulaire de création de publication (fig. 7.1).

Puis nous allons ajouter un champ qui permettra d'uploader un fichier. Dans `create.ejs`, ajouter le champ suivant :

```
<form action="/posts/store" method="POST"
  enctype="multipart/form-data">
  ...
  <div class="control-group">
    <div class="form-group floating-label-form-group
      controls">
      <label>Image</label>
      <input type="file" class="form-control" id="image"
        name="image">
    </div>
```

```
</div>  
<br>  
<div class="form-group">
```

Explication du code

```
<form action="/posts/store" method="POST"  
  enctype="multipart/form-data">
```

enctype="multipart/form-data" permet au navigateur de savoir que le formulaire contient du multimédia.

Le navigateur cryptera alors le multimédia avant de l'envoyer au serveur.

Dans **index.js**, dans le gestionnaire de requêtes pour « /posts/store », ajoutez :

```
app.post('/posts/store', (req,res)=>{  
  let image = req.files.image;  
  
  image.mv(path.resolve(__dirname, 'public/img', image.name),  
    async (error)=>{  
      await BlogPost.create(req.body)  
      res.redirect('/')  
    })  
})
```

Nous créons d'abord un raccourci vers `req.files.image` avec `image`.

L'objet `req.files.image` contient certaines propriétés comme `mv` - une fonction pour déplacer le fichier ailleurs sur votre serveur et votre nom. (Voir <https://www.npmjs.com/package/express-fileupload> pour la liste complète de ce qu'il contient.)

image.mv déplace le fichier téléchargé vers le répertoire public/img avec son nom à partir de l'image.nom. Lorsque cela est fait, nous procédons à la création de poste comme ce que nous avons fait avant. Notez le positionnement de l'instruction `async`. Uniquement dans le cadre où nous utilisons **await**, **async** devrait être déclaré.

Exécuter votre application

Lorsque vous exécutez votre application, essayez de créer un article de blog et de télécharger une image. Vous devriez pouvoir voir l'image téléchargée dans public/img de votre répertoire local.

Enregistrement des images téléchargées dans la base de données

Dans **BlogPost.js**, ajoutez une variable image de type String pour stocker le chemin du fichier du image comme indiqué :

```
const BlogPostSchema = new Schema({
  title: String,
  body: String,
  username: String,
  datePosted:{
    type: Date,
    default: new Date()
  },
  image: String
});
```

Dans index.js, spécifiez le chemin complet du fichier image vers l'attribut d'image BlogPost dans **create()** comme indiqué ci-dessous :

```
app.post('/posts/store', (req,res)=>{
```

```
let image = req.files.image;
image.mv(path.resolve(__dirname, 'public/img', image.name),
  async(error)=>{
    await BlogPost.create({
      ...req.body,
      image: '/img/' + image.name
    })
    res.redirect('/')
  })
})
```

Lorsque vous exécutez votre application maintenant, le nouveau billet de blog contiendra le chemin du fichier image.

Pour afficher l'image dans le post, dans **post.ejs**, modifiez le code en dur pour le chemin du fichier de :

```
<header class="masthead" style="background-image:
url('img/post-bg.jpg')">
```

à:

```
<header class="masthead" style="background-image:
url('<%= blogpost.image %>')">
```

Désormais, lorsque vous exécutez votre application et accédez à une page d'article de blog spécifique avec une image, il devrait montrer l'image dans la vue du header.

Résumé

Dans ce chapitre, nous avons exploré comment télécharger des images pour chaque article de blog et les stocker sur le serveur.

Chap. 8 Introduction à Express Middleware

Les middleware sont des fonctions qu'Express exécute au milieu après une demande entrante qui produit ensuite une sortie qui peut être soit la sortie finale ou être utilisé par le prochain middleware.

On peut avoir plus d'un middleware et ils s'exécuteront dans l'ordre où ils seront déclarés.

Dans le processus, les middlewares peuvent apporter des modifications aux objets **request** et **response**.

Nous avons déjà en fait utilisé des middlewares. Dans **index.js**, quand on a fait **app.use(...)**, nous appliquons en fait des middlewares à Express.

Nous avons par exemple déclaré les middlewares suivants :

```
app.use(express.static('public'))
app.use(bodyParser.json())
app.use(bodyParser.urlencoded({extended:true}))
app.use(fileUpload())
```

La fonction **use** enregistre un middleware avec notre application Express.

Ainsi, lorsqu'un navigateur fait une requête à une page par exemple, Express exécutera toutes les instructions « use » de manière séquentielle avant de traiter la demande.

Avec **app.use(fileUpload())**, le middleware **fileUpload** modifie la requête et lui ajoute la propriété **request.files**.

Sans le téléchargement de fichier middleware, télécharger un fichier et le récupérer serait beaucoup plus difficile !

Middleware personnalisé

Vous pouvez même créer votre propre middleware personnalisé. Par exemple, dans **index.js**, ajoutez le code suivant pour créer un middleware personnalisé :

```
const customMiddleware = (req,res,next)=>{  
  console.log('Middleware personnalisé appelé')  
  next()  
}  
app.use(customMiddleware)
```

Désormais, chaque fois que vous actualisez votre application, le message « **Middleware personnalisé appelé** » sera enregistré dans la console.

next() indique à Express que le middleware est terminé et qu'Express doit appeler la fonction middleware suivante.

Si vous supprimez **next()** et accédez à votre application dans le navigateur, l'application se bloquera car vous n'avez pas demandé à Express de passer à la fonction middleware suivante.

Tous les middlewares appelés par `app.use` appellent **next()**

Enregistrement du middleware de validation

Un bon cas d'utilisation d'un middleware est la validation de formulaire, où nous vérifions la validité des valeurs des champs du formulaire (champ non vide etc.).

Sans formulaire de validation, notre application plante actuellement si nous soumettons un nouveau formulaire de publication vide.

Nous obtenons une erreur `TypeError` : Impossible de lire la propriété 'image' de null car la propriété image ne peut pas être nulle.

Avec un middleware de validation, Express vérifie si les données sont remplies dans les champs est valide avant d'envoyer la requête au serveur.

Créer la validation middleware avec le code ci-dessous :

```
const validateMiddleware = (req,res,next)=>{  
  if(req.files == null || req.body.title == '' ||  
    req.body.body == ''){  
    return res.redirect('/posts/new')  
  }  
  next()  
}
```

Le middleware **validateMiddleware** vérifie simplement si l'un des champs du formulaire est null (ce qui signifie qu'ils ne sont pas saisis par l'utilisateur) et si oui, rediriger les faire revenir à la page de création de publication.

Si nous appliquons le middleware à notre application en utilisant **app.use(validateMiddleware)**, ce middleware sera exécuté pour toute demande alors que nous voulons seulement qu'elle soit exécutée pour la demande de création de postes.

Ainsi, pour appliquer un middleware à des requêtes d'url spécifiques, nous faisons :

```
app.use('/posts/store',validateMiddleware)
```

C'est-à-dire que si Express voit une demande de l'URL donnée « /posts/store », alors exécute le middleware **validateMiddleware**.

Remarque : assurez-vous que les éléments ci-dessus l'instruction est après **app.use(fileUpload())** puisque nous dépendons de l'objet req ayant la propriété files.

Lorsque vous exécutez votre application et essayez de soumettre un formulaire de création de publication avec l'un des champs manquants, vous serez redirigé vers la page du formulaire. Seulement quand vous avez tous les champs remplis votre envoi sera réussi.

Résumé

Nous avons vu les middlewares dans Express et comment ils ajoutent une fonctionnalité à notre application en ajoutant des fonctionnalités à la demande et à la réponse objets. Nous avons utilisé le middleware de validation pour valider les formulaires.

Chap. 9 : Refactorisation avec MVC

Jusqu'à présent, nous avons mis tout notre code dans `index.js`. Par exemple, tous les gestionnaires de requêtes **`app.get`**, **`app.post`** sont dans le même fichier.

Ce n'est pas la meilleure approche lorsque notre application se développera ou si nous construisons une application plus complexe car **`index.js`** devient souvent trop compliqué et ingérable.

Ainsi, nous allons refactoriser notre code en utilisant un modèle commun appelé Model-View-Controller ou Modèle MVC.

C'est-à-dire que nous allons diviser notre application en trois parties interconnectées, **Model**, **View** et **Controller** :

- **Model** représente la structure des données, le format et les contraintes avec lesquelles elles sont stockées. En substance, c'est la partie base de données de l'application.

Comme vous l'avez peut-être déjà remarqué, nous l'avons déjà dans notre dossier **`models`** stockant actuellement le modèle `BlogPost`.

- **View** est ce qui est présenté à l'utilisateur. Les **Views** utilisent le modèle et présentent les données de la manière souhaitée par l'utilisateur.

Depuis les views, l'utilisateur peut modifier la présentation des données qui lui sont présentées.

Dans notre application, **View** compose de façon statique ou dynamique les pages rendues aux utilisateurs. Les pages sont stockées dans un dossier de vues stockées dans divers fichiers EJS pour rendre des sites Web HTML statiques et dynamiques.

- Enfin, nous avons **Controller** qui contrôle les demandes de l'utilisateur puis génère une réponse appropriée renvoyée à l'utilisateur. c'est-à-dire qu'un utilisateur interagit avec une **View** qui génère la requête appropriée qui est gérée par le **Controller** qui restitue ensuite la **View** appropriée avec les données du **Model** comme réponse.

Nous avons déjà la couche Model et View. Dans ce chapitre, nous allons refactoriser notre code pour créer une couche Controller en créant un dossier Controller qui stockera nos gestionnaires de requêtes.

Alors, créez un nouveau dossier **controllers** et créez-y un nouveau fichier **newPost.js**.

Ce fichier contiendra le contrôleur traitant la demande de l'utilisateur de créer un nouveau billet de blog.

Dans newPost.js, insérez le code suivant :

```
module.exports = (req, res) =>{  
  res.render('create')  
}
```

Dans **index.js**, nous remplaçons ensuite le gestionnaire de requêtes ci-dessous :

```
app.get('/posts/new', (req, res) =>{  
  res.render('create')  
})
```

par:

```
const newPostController =  
  require('./controllers/newPost')  
...  
app.get('/posts/new', newPostController)
```

C'est-à-dire que nous déplaçons la fonction de gestionnaire de requêtes existante vers un contrôleur séparé , ici le fichier **newPost.js**.

Cela aide à rendre `index.js` plus petit et à fournir une meilleure organisation en séparant les détails du contrôleur dans leur propre fichier.

Exercice

Essayez par vous-même de créer des fichiers de contrôleur individuels pour les autres fonctions du gestionnaire de requêtes.

Les avez-vous essayés ? Voyons cela ensemble maintenant. Pour plus de simplicité dans notre projet et pour nous permettre de nous concentrer sur l'essentiel, nous supprimerons About, Contact et SAmple Post , car il ne s'agit que de pages HTML statiques standards.

Donc, dans **`index.js`**, supprimez :

```
app.get('/about', (req, res) => {  
  res.render('about');  
})  
app.get('/contact', (req, res) => {  
  res.render('contact');  
})  
app.get('/post', (req, res) => {  
  res.render('post')  
})
```

Nous devons aussi enlever leurs liens qui sont dans `navbar.ejs` :

Nous devons donc remanier la page d'accueil, stocker la publication et obtenir une publication individuelle avec le gestionnaire de demandes.

Dans le dossier des contrôleurs, créez `home.js`, `storePost.js` et `getPost.js` comme indiqué ci-dessous :

`home.js`

```
const BlogPost = require('../models/BlogPost.js')
module.exports = async (req, res) =>{
  const blogposts = await BlogPost.find({})
  res.render('index', {
    blogposts
  });
}
```

Notez que nous devons importer le modèle `BlogPost` avec le **`require`** qui monte un répertoire (`..`).

`getPost.js`

```
const BlogPost = require('../models/BlogPost.js')

module.exports = async (req, res) =>{
  const blogpost = await BlogPost.findById(req.params.id)
  console.log(blogpost)
  res.render('post', {
    blogpost
  });
}
```

storePost.js

```
const BlogPost = require('../models/BlogPost.js')
const path = require('path')

module.exports = (req, res) => {
  let image = req.files.image;
  image.mv(path.resolve(__dirname, '..', 'public/img', image.name), async
(error) => {
    await BlogPost.create({
      ...req.body,
      image: '/img/' + image.name
    })
    res.redirect('/')
  })
}
```

Notez que nous devons importer path. Nous devons également ajouter « .. » à path.resolve car nous devons remonter d'un dossier avant de faire référence à public/img.

Dans **index.js**, importez les contrôleurs suivants que nous venons de créer :

```
const homeController = require('./controllers/home')
const storePostController = require('./controllers/storePost')
const getPostController = require('./controllers/getPost')
```

Apportez les modifications suivantes aux gestionnaires de requêtes pour appeler les fonctions du nouveau contrôleur:

```
app.get('/', homeController)
app.get('/post/:id', getPostController)
app.post('/posts/store', storePostController)
```

Parce que **index.js** ne nécessite plus **path** et **BlogPost**, nous pouvons les supprimer.

```
const path = require('path')  
const BlogPost = require('../models/BlogPost.js')
```

Voyez à quel point **index.js** est maintenant plus léger et le code est mieux organisé en faisant abstraction vers une couche de contrôleurs.

Couche de validation de la refactorisation

Nous pouvons aller plus loin et placer notre middleware de validation dans son propre dossier.

Créez un répertoire **middleware** et créez-y un nouveau fichier **validationMiddleware.js**.

Couper et coller la fonction **validationMiddleware** de **index.js** dans **validationMiddleware.js** comme indiqué ci-dessous :

```
module.exports = (req, res, next) => {  
  if (req.files == null || req.body.title == null ||  
    req.body.title == null) {  
    return res.redirect('/posts/new')  
  }  
  next()  
}
```

De retour dans **index.js**, importez le middleware avec :

```
const validateMiddleware =  
  require("../middleware/validateMiddleware");
```

Si vous exécutez votre application maintenant, la validation du formulaire devrait avoir lieu comme avant.
Avec cela, nous avons refactorisé notre code middleware.

Dans ce chapitre, nous avons refactorisé notre code de projet dans le Model - View - Controller qui rend notre code plus gérable et organisé.

Nous avons particulièrement implémenté des contrôleurs et extrait du code loin d'`index.js` l'empêchant de devenir désordonné.

Chap. 10 Enregistrement d'un utilisateur

Jusqu'à présent, nous avons créé des articles de blog sans toutefois leur attribuer un nom d'utilisateur.

En réalité, un utilisateur doit d'abord enregistrer un compte utilisateur et se connecter avant de créer des articles de blog.

Les messages seront ensuite attribués à cet utilisateur.

Dans ce chapitre, nous mettrons en œuvre l'enregistrement des utilisateurs.

Nous créons d'abord la vue enregistrement utilisateur. Dans le dossier **'views'**, créez un nouveau fichier **'register.ejs'**.

Puisque nous allons avoir un formulaire, nous utiliserons le code dans `create.ejs` qui contient déjà un formulaire et copiez son contenu dans `register.ejs`.

Dans `register.ejs`, remplacez l'en-tête par "Enregistrer un nouveau compte"

```
<div class="page-heading">
  <h1>Enregistrement Nouvel Utilisateur</h1>
</div>
```

Et aussi, changer la destination de l'action dans la balise `form` :

```
<form action="/users/register" ...>
```

Dans le champ **title**, changer le **username** comme suit :

```
<div class="control-group">
  <div class="form-group floating-label-form-group
controls">
    <label>Nom Utilisateur</label>
    <input type="text" class="form-control"
placeholder="Nom Utilisateur"
id="username" name="username">
  </div>
</div>
```

Supprimez le code pour la description car il s'agit d'un champ de zone de texte.

Dupliquer le code pour le champ nom d'utilisateur et renommez-le comme mot de passe pour le champ mot de passe comme indiqué ci-dessous.

Notez que le type doit être un mot de passe pour masquer les caractères étant saisis.

Pour l'instant, nous n'incluons pas d'autres champs pour simplifier notre projet. Ainsi, nous supprimerons le champ image.

Ensuite, renommez le bouton Soumettre en S'inscrire.

Dans le dossier **Controllers**, créer le fichier **newUser.js** avec le code suivant :

```
module.exports = (req, res) =>{
  res.render('register') // renvoie register.ejs
}
```

Importer le nouveau Controller dans **index.js** :

```
const newUserController =
require('./controllers/newUser')
```

Puis on lui fournit sa route :

```
app.get('/auth/register', newUserController)
```

Maintenant, ajoutez le lien « Nouvel utilisateur » dans la barre de navigation afin que l'on puisse cliquer pour S'inscrire:

Dans **views/layouts/navbar.ejs**, ajoutez :

```
<li class="nav-item">
  <a class="nav-link" href="/auth/register">Nouvel
  Utilisateur</a>
</li>
```

Ce qui précède appellera alors `newUserController` pour rendre `register.ejs`. Si vous lancez votre application maintenant et cliquez sur « Nouvel utilisateur » dans la barre de navigation, vous serez amené à la vue du registre (Fournissez une nouvelle image pour la page d'enregistrement).

Model Utilisateur

Pour l'enregistrement des utilisateurs, nous avons besoin d'un modèle d'utilisateur pour représenter notre collection d'utilisateurs tout comme ce que nous avons pour `BlogPost`. Dans le dossier `models`, créez un nouveau fichier **User.js** et copiez/éditez le contenu de **BlogPost.js** pour créer le schéma utilisateur :

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema;
const UserSchema = new Schema({
  username: String,
  password: String
});
// export model
const User = mongoose.model('User', UserSchema);
module.exports = User
```

Notre modèle est désormais capable d'enregistrer des utilisateurs. Configurons l'itinéraire qui va être appelé lorsqu'un utilisateur essaie d'enregistrer un compte.

Allez sur `index.js` et enregistrez un nouveau parcours avec :

```
...  
const storeUserController =  
require('./controllers/storeUser')  
...  
app.post('/users/register', storeUserController)  
...
```

Parce que nous avons enregistré la route en tant que `/users/register`, dans `register.ejs`, spécifiez ce même itinéraire dans l'action du formulaire du formulaire d'inscription de l'utilisateur :

```
<form action="/users/register" method="POST"  
enctype="multipart/form-data">
```

Gérer l'enregistrement de l'utilisateur

Nous créons maintenant **storeUser.js** pour que notre **storeUserController** gère l'enregistrement de l'utilisateur. Dans le dossier **controllers**, créez un nouveau fichier **storeUser.js** avec le code suivant :

```
const User = require('../models/User.js')  
const path = require('path')  
module.exports = (req, res) => {  
  User.create(req.body, (error, user) => {  
    res.redirect('/')  
  })  
}
```

Si vous exécutez votre application maintenant et créez un nouvel utilisateur avec le nouveau formulaire utilisateur, il créera un nouveau document utilisateur dans la base de données sous la collection utilisateurs.

Notez que votre mot de passe est actuellement en texte brut, ce qui est très peu sûr. Nous devons hacher le mot de passe avant de le stocker dans MongoDB.

Cryptage du mot de passe

Nous utiliserons un hook de modèle mongoose pour crypter un mot de passe avant de le stocker.

Un hook est comme un objet intermédiaire. Nous installons d'abord un paquet appelé bcrypt (<https://www.npmjs.com/package/bcrypt>) pour nous aider à hacher les mots de passe. Installez-le utilisant:

```
npm i --save bcrypt
```

Maintenant dans **/models/User.js**, ajoutez les codes ci-dessous :

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema;
const bcrypt = require('bcrypt')
```

et avant l'export :

```
UserSchema.pre('save', function(next){
  const user = this
  bcrypt.hash(user.password, 10, (error, hash) => {
    user.password = hash
  })
  next()
})
```

Explication du code

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

D'abord nous importons le package bcrypt dans User.js :

```
UserSchema.pre('save', function(next) {  
  ...  
})
```

Ensuite, avec `UserSchema.pre(' save '... ,` nous disons à Mongoose qu'avant d'enregistrer n'importe quel enregistrement dans le schéma Users ou la collection Users, exécutez la fonction passée dans le 2ème argument.

Cela nous permet de modifier les données utilisateur avant de les enregistrer dans la base de données.

*Notez que nous devons spécifier `function(next)` au lieu d'utiliser la forme courte lambda d'une fonction, c'est-à-dire `next => {...}` .

```
UserSchema.pre('save', function(next) {  
  const user = this  
  bcrypt.hash(user.password, 10, (error, hash) => {  
    user.password = hash  
    next()  
  })  
})
```

Dans la fonction, nous obtenons d'abord l'utilisateur enregistré avec `const user = this`.

mongoose rend le UserSchema disponible via ceci.

Nous procédons ensuite à l'appel de `bcrypt.hash` dont le premier argument prend le mot de passe à hacher.

Le deuxième argument spécifie le nombre de tours de hachage à effectuer. Par exemple, nous avons spécifié 10 ce qui signifie que le mot de passe sera crypté 10 fois.

Vous pouvez bien sûr spécifier 100. Plus il y a de tours de hachage, plus c'est sûr mais plus le processus est lent aussi.

Le troisième argument est la fonction qui est appelée lorsque le hachage est terminé.

`user.password = hash` remplace le mot de passe d'origine par sa version cryptée.

Nous appelons ensuite `next()` pour que mongoose puisse continuer à créer les données utilisateur.

Essayez maintenant de créer un nouvel utilisateur dans votre application et dans la base de données, vous devriez voir un mot de passe crypté.

Validation par Mongoose

Nous avons utilisé mongoose pour nous aider à crypter notre mot de passe.

Une autre caractéristique importante d'un formulaire est de valider les données saisies par l'utilisateur avant de les stocker dans la base de données.

Lors de la création de nouveaux documents, mongoose peut nous aider à valider les données avant de les enregistrer dans la base de données.

Nous le faisons dans le schéma. Par exemple, `UserSchema` est actuellement :

```
...  
const UserSchema = new Schema({  
  username: String,  
  password: String  
});  
...
```

Supposons que nous voulions nous assurer que le nom d'utilisateur est obligatoire et unique dans la base de données, nous faisons :

```
const UserSchema = new Schema({
  username: {
    type: String,
    required: true,
    unique: true
  }, password: {
    type: String,
    required: true
  }
});
```

C'est-à-dire que nous passons un objet de configuration pour chaque champ et spécifions des règles de validation dedans.

Dans l'objet config, nous avons le champ obligatoire qui spécifie que le Champ est requis.

Nous appliquons cela à la fois au champ nom d'utilisateur et mot de passe.

Pour le nom d'utilisateur, nous définissons unique sur true, Mongoose vérifiera donc que ce nom d'utilisateur doit être unique pour cet enregistrement avant de l'enregistrer dans la base de données.

Si nous exécutons notre application et enregistrons un utilisateur sans ces champs ou si le nom d'utilisateur n'est pas unique, nous verrons qu'il n'y a pas d'utilisateur nouvellement créé dans la base de données.

Mais un problème que nous avons maintenant est que lorsqu'un utilisateur n'est pas créé en raison d'un échec dans le champ validation, aucune notification n'est fournie à l'utilisateur.

C'est-à-dire que l'on pourrait supposer que l'utilisateur a été ajouté alors qu'en réalité il ne l'a pas été. Plus tard, nous verrons comment notifier les erreurs de validation à l'utilisateur sur la page.

Mais pour l'instant, on envoie les erreurs à la console.

Pour enregistrer les erreurs, accédez simplement à `storeUser.js` et ajoutez `console.log(error)` comme cela :

```
const User = require('../models/User.js')
const path = require('path')
module.exports = (req,res)=>{
  User.create(req.body, (error, user) => {
    console.log(error)
    res.redirect('/')
  })
}
```

Maintenant, si nous essayons d'ajouter un nom d'utilisateur qui existe déjà, nous obtenons l'erreur suivante :

'E11000 duplicate key error collection: my_database.users index: username_1 dup key: { : "..."}'

Avoir des erreurs enregistrées dans la console n'est évidemment pas idéal pour les personnes utilisant notre application qui ne peut pas (et ne veut pas) faire référence aux logs !

Nous afficherons plus tard la notification d'erreur dans la page elle-même. Mais pour l'instant, quand il y a une erreur, nous vous redirigerons vers le formulaire d'inscription des utilisateurs avec le code suivant :

```
...
module.exports = (req,res)=>{
  User.create(req.body, (error, user) => {
    if(error){
      return res.redirect('/auth/register')
    }
    res.redirect('/')
  })
}
```

Processus de connexion utilisateur

Après avoir créé notre page d'enregistrement des utilisateurs, nous créons ensuite une page de connexion.

Dans le dossier **views**, créez un nouveau fichier **login.ejs**.

login.ejs sera similaire à register.ejs donc

copiez le code de register.ejs dans login.ejs. Les étapes suivantes doivent vous être familières.

Dans login.js, changez l'en-tête de la page et le libellé du bouton en « Connexion » :

```
<div class="page-heading">
<h1>Connexion</h1>
</div>
...
<div class="form-group">
<button type="submit" class="btn
btn-primary">Login</button>
</div>
```

Ensuite dans l'action du formulaire, passez de
<form action="/users/register" ... à:

```
<form action="/users/login"...
```

et dans le dossier controllers, créez un nouveau fichier login.js avec le code suivant :

```
module.exports = (req, res) =>{
  res.render('login')
}
```

Puis dans index.js, importer le contrôleur de login :

```
const loginController = require('./controllers/login')
```

Et utiliser la route ainsi déclarée :

```
app.get('/auth/login', loginController);
```

Enfin, nous ajoutons login à la navbar en allant dans view/layouts/navbar.ejs :

```
<li class="nav-item">
  <a class="nav-link" href="/auth/login">Login</a>
</li>
```

Notez que l'élément Login a été placé avant l'élément Nouvel utilisateur car cela adhère à un principe de conception d'interface utilisateur où l'élément le plus fréquemment utilisé doit être placé en premier, et l'on est susceptible d'utiliser l'élément de connexion plus de enregistrer un nouvel utilisateur.

Processus de connexion

Dans cette section, nous implémentons le processus de connexion pour comparer le mot de passe et le rediriger vers la page d'accueil si la connexion est réussie.

D'abord dans le dossier des contrôleurs, créez un nouveau fichier loginUser.js avec le code suivant :

```
const bcrypt = require('bcrypt')
const User = require('../models/User')
module.exports = (req, res) =>{
  const { username, password } = req.body;
  User.findOne({username:username}, (error,user) => {
    if (user){
      bcrypt.compare(password, user.password, (error, same) =>{
        if(same){ // if passwords match
          // store user session, will talk about it later
          res.redirect('/')
        }
        else{
```

```
res.redirect('/auth/login')
  })
}
else{
res.redirect('/auth/login')
}
})
}
```

Explication du code

```
const bcrypt = require('bcrypt')
const User = require('../models/User')
module.exports = (req, res) =>{
  const { username, password } = req.body;
```

Nous importons le package bcrypt et le modèle User. Ensuite, nous extrayons le nom d'utilisateur et le mot de passe du formulaire de connexion utilisateur avec req.body.

```
User.findOne({username:username}, (error,user) => {
  if (user){
    ...
  }
  else{
res.redirect('/auth/login')
  }
})
```

Nous utilisons ensuite `User.findOne` pour essayer de trouver un seul utilisateur avec le Nom d'utilisateur entré. Si un tel utilisateur existe, nous procédons à la comparaison des mots de passe. Si l'utilisateur n'existe pas, nous retournons à la page de connexion.

Pour comparer les mots de passe, nous utilisons `bcrypt.compare` pour comparer le mot de passe entré avec le mot de passe utilisateur haché récupéré dans notre base de données.

Noter que nous utilisons `bcrypt.compare` au lieu d'un contrôle d'égalité, par ex. `===`. C'est pour nous protéger d'une astuce de piratage appelée attaque temporelle (timing attack).

Si les mots de passe correspondent, nous redirigeons vers la page d'accueil où vous pouvez voir la liste des articles de blog. Si les mots de passe ne correspondent pas, nous redirigeons vers la page login.

loginUserController

Pour appliquer notre **loginUserController**, dans `index.js`, importez-le en utilisant :

```
const loginUserController =  
require('./controllers/loginUser')
```

et créez la route :

```
app.post('/users/login', loginUserController)
```

L'itinéraire doit être le même que celui de l'action du formulaire dans `login.ejs`.

Essayez maintenant d'exécuter votre application et connectez-vous. Vous devriez être redirigé vers la page de connexion si la connexion échoue et dirigé vers la page d'accueil si la connexion est réussie.

Résumé

Nous avons créé un formulaire d'inscription d'utilisateur pour ajouter des utilisateurs à notre collection d'utilisateurs dans la base de données.

Le package bcrypt a été utilisé pour hacher les mots de passe des utilisateurs avant de les stocker.

Nous avons utilisé la validation Mongoose pour valider les données saisies par l'utilisateur.

Nous avons ensuite mis en œuvre le processus de connexion pour comparer les mots de passe et autoriser la connexion lorsque l'authentification est réussie.

Chap. 11 Authentification de l'utilisateur avec les sessions Express

Les sessions sont la façon dont nous gardons l'utilisateur connecté à notre application Web en gardant ses informations dans le navigateur.

Chaque fois qu'un utilisateur fait une demande, les informations de cet utilisateur sont renvoyées au serveur.

Le serveur sait ainsi quel utilisateur fait cette demande et s'il est connecté ou déconnecté. Les informations conservées sur le navigateur de l'utilisateur sont appelées **cookies**.

Pour implémenter les sessions Express, nous installons un package middleware appelé express-session (<https://github.com/expressjs/session>):

```
npm install --save express-session
```

Ensuite, importez ce middleware dans index.js avec :

```
const expressSession = require('express-session');
```

Et ajoutez ceci :

```
app.use(expressSession({  
  secret: 'keyboard cat',  
  resave: true,  
  saveUninitialized: true  
}))
```

Dans le code ci-dessus, nous enregistrons le middleware `expressSession` dans notre application et transmettons un objet de configuration avec une valeur à la propriété `secret`.

La chaîne secrète est utilisée par le package `express-session` pour signer et crypter l'ID de session cookie partagé avec le navigateur. Vous pouvez bien sûr fournir votre propre chaîne secrète.

Maintenant, allez dans Chrome, actualisez votre application et dans Developer Tools, accédez à l'onglet 'application', et sous Cookies, domaine `localhost`, vous verrez que nous avons un **`connect.sid`** qui contient une valeur hachée.

Cette valeur est l'ID de session hachée échangée avec le serveur et le navigateur de l'utilisateur nous indiquant quel utilisateur est actuellement connecté.

Si vous commentez le code d'`express-session` ci-dessus, effacez les cookies et actualisez l'app, vous ne verrez pas ce cookie.

Implémentation de sessions utilisateur

Pour implémenter la session utilisateur, dans **loginUser.js**, ajoutez la ligne suivante :

```
const bcrypt = require('bcrypt')
const User = require('../models/User')
module.exports = (req, res) =>{
  const { username, password } = req.body;
  User.findOne({username:username}, (error,user) => {
    if (user){
      bcrypt.compare(password, user.password, (error, same) =>{
        if(same){
          req.session.userId = user._id
          res.redirect('/')
        }
        else{
          res.redirect('/auth/login')
        }
      })
    }
    else{
      res.redirect('/auth/login')
    }
  })
}
```

Nous attribuons le `user_id` à la session. Le package **session** enregistre ces données sur le navigateur de l'utilisateur afin que chaque fois que l'utilisateur fait une demande, ce cookie sera renvoyé au serveur avec l'identifiant authentifié.

C'est ainsi que nous savons si un utilisateur est connecté. Pour voir exactement ce qu'il y a dans un objet de session, allez sur **home.js** et ajoutez la ligne suivante:

```
const BlogPost = require('../models/BlogPost.js')
module.exports = async (req, res) =>{
  const blogposts = await BlogPost.find({})
  console.log(req.session)
  res.render('index',{
    blogposts
  });
}
```

Lorsque vous vous connectez à votre application et accédez à la page d'accueil, vous verrez que la session a des données de cookie avec des informations d'ID utilisateur, par exemple :

```
Session {
  cookie:
    { path: '/',
      _expires: null,
      originalMaxAge: null,
      httpOnly: true },
  userId: '60d740b3da86f51622337771' }
```

userId sera partagé au sein du serveur et du navigateur. Ainsi, dans à demande, le serveur saura si l'utilisateur est connecté ou non.

Pour mettre en œuvre la vérification d'un identifiant de session avant d'autoriser un utilisateur à créer un article de blog, dans **newPost.js**, implémentez les éléments suivants :

```
module.exports = (req, res) => {
  if (req.session.userId) {
    return res.render("create");
  }
  res.redirect('/auth/login')
}
```

Nous vérifions si la session contient un identifiant utilisateur. Si c'est le cas, affichez la création d'une page de publication. Si ce n'est pas le cas, c'est redirigé vers la page de connexion.

Protection des pages avec un middleware d'authentification

Maintenant que nous avons l'authentification, nous allons protéger les pages dont nous ne voulons pas qu'elles soient manipulées par des utilisateurs non connectés.

Par exemple, nous voulons uniquement des utilisateurs connectés pour accéder à la page « publier un nouveau post ».

Tout d'abord, nous créons un middleware personnalisé dans **/middleware/authMiddleware.js** avec le code suivant :

```
const User = require('../models/User')
module.exports = (req, res, next) => {
  User.findById(req.session.userId, (error, user) => {
    if (error || !user)
      return res.redirect('/')
    next()
  })
}
```

Dans celui-ci, nous récupérons l'utilisateur de la base de données avec **User.findById(req.session.userId ...**

Nous vérifions ensuite si l'utilisateur est récupéré avec succès ou si l'utilisateur n'existe pas, nous renvoyons la page d'accueil.

Si l'utilisateur est un utilisateur valide, nous autorisons la demande et continuons avec next().

Maintenant, pour appliquer ce middleware, dans index.js, importez **authMiddleware** avec :

```
const authMiddleware =
require('../middleware/authMiddleware');
```

et dans la définition de route existante pour `/posts/new`, transmettez le middleware dans :

```
app.get('/posts/new', authMiddleware, newPostController)
```

Vous pouvez voir ce qui précède comme un pipeline allant de gauche à droite où nous appelons **authMiddleware** avant d'appeler **newPostController**.

Nous appliquons la même chose pour la publication des posts :

```
app.post('/posts/store', authMiddleware,  
storePostController)
```

Avec cela, un utilisateur non authentifié ne peut pas accéder au nouveau formulaire de publication et soumettre un article de blog.

Pour tester votre application, supprimez d'abord la session existante du navigateur. Et lorsque vous essayez de créer un nouveau message avant de vous connecter, vous devriez être redirigé vers la page d'accueil.

Empêcher la connexion/l'enregistrement si vous êtes connecté

Actuellement, si un utilisateur est connecté, il peut toujours visiter la page de connexion ou de nouvel utilisateur ce qui ne devrait pas être le cas.

Tout comme nous autorisons seulement les utilisateurs connectés pour créer des publications, nous allons créer un middleware pour vérifier si l'utilisateur est authentifié, et si c'est le cas, l'empêcher d'accéder au login/new utilisateur.

Dans le dossier middleware, créez le fichier **redirectIfAuthenticatedMiddleware.js** avec le code suivant :

```
module.exports = (req, res, next) => {  
  if (req.session.userId) {  
    return res.redirect('/') // if user logged in,  
    redirect to home page  
  }  
  next()  
}
```

Le code ci-dessus devrait vous être familier. Ensuite dans index.js, importez **redirectIfAuthenticatedMiddleware** avec :

```
const redirectIfAuthenticatedMiddleware =  
require('./middleware/redirectIfAuthenticatedMiddleware'  
)
```

Ensuite, appliquez-le aux quatre routes suivantes :

```
app.get('/auth/register',  
  redirectIfAuthenticatedMiddleware, newUserController)  
app.post('/users/register',  
  redirectIfAuthenticatedMiddleware, storeUserController)  
app.get('/auth/login', redirectIfAuthenticatedMiddleware,  
  loginController)  
app.post('/users/login', redirectIfAuthenticatedMiddleware,  
  , loginUserController)
```

Maintenant, lorsque vous êtes connecté, vous serez redirigé vers la page d'accueil lorsque vous cliquez sur Connexion ou Nouvel utilisateur.

Affichage conditionnel des liens NewPost, Login et New User

Suivant la section ci-dessus, au lieu de rediriger vers la page d'accueil chaque fois qu'un utilisateur connecté clique sur Connexion ou " nouvel utilisateur " , nous devrions masquer le nouvel utilisateur et les liens de connexion si un utilisateur est déjà connecté.

Pour ce faire, ajoutez dans index.js :

```
global.loggedIn = null;  
app.use("*", (req, res, next) => {  
  loggedIn = req.session.userId;  
  next()  
});
```

Explication du code

```
global.loggedIn = null;
```

Nous déclarons d'abord une variable globale **loggedIn** qui sera accessible depuis tous nos fichiers EJS.

Parce que la barre de navigation existe dans tous nos fichiers EJS, ils doivent chacun accéder au login pour modifier la barre de navigation.

Avec `app.use("*", (req, res, next) => ...`, nous spécifions avec le symbole (wildcard) `*`, que sur toutes les requêtes, ce middleware doit être exécuté.

Dans celui-ci, nous assignons **loggedIn** à `req.session.userId`.

Maintenant, allez dans **views/layout/navbar.ejs** et utilisez une instruction if pour Newpost, login et register.

```
<div class="collapse navbar-collapse" id="navbarResponsive">
  <ul class="navbar-nav ml-auto">
    <li class="nav-item">
      <a class="nav-link" href="/">Home</a>
    </li>
    <% if (loggedIn) { %>
      <li class="nav-item">
        <a class="nav-link" href="/posts/new">New
Post</a>
      </li>
    <% } %>
    <% if (!loggedIn) { %>
      <li class="nav-item">
        <a class="nav-link"
href="/auth/login">Login</a>
      </li>
      <li class="nav-item">
        <a class="nav-link"
href="/auth/register">New User</a>
      </li>
    <% } %>
  </ul>
</div>
```

Si login est nul, nous affichons les liens login/nouvel utilisateur et masquons le lien « new post ». Si le login a une valeur, c'est-à-dire l'identifiant de session, nous masquons la connexion et « New User » et afficher le lien « new post ». Et parce que toutes les pages incluent navbar.ejs, il s'appliquera à toutes les pages.

Déconnexion de l'utilisateur

Actuellement, un utilisateur connecté ne peut pas se déconnecter. Créons un itinéraire pour qu'un utilisateur puisse se déconnecter.

Dans la barre de navigation, ajoutez les éléments suivants :

```
<li class="nav-item">
  <a class="nav-link" href="/posts/new">New Post</a>
</li>
<li class="nav-item">
  <a class="nav-link" href="/auth/logout">Log out</a>
</li>
<% } %>
```

Le code ci-dessus affiche le lien de déconnexion uniquement si un utilisateur est connecté.

Nous avons également spécifié la route pour le lien de déconnexion comme /auth/logout.

Ensuite, accédez à index.js et créez la route et son contrôleur. Créer la déconnexion dans le contrôleur dans un nouveau fichier **/controllers/logout.js** avec le code :

```
module.exports = (req, res) => {
  req.session.destroy(() => {
    res.redirect('/')
  })
}
```

Avec **req.session.destroy()**, nous détruisons toutes les données de session, y compris la session identifiant d'utilisateur, nous redirigeons ensuite vers la page d'accueil.

Nous appliquons ensuite le contrôleur de déconnexion à index.js avec :

```
const logoutController = require('./controllers/logout')
```

et on applique le controller sur cette route :

```
app.get('/auth/logout', logoutController)
```

Si vous exécutez votre application maintenant, vous pourrez vous déconnecter avec toutes les données de session supprimées.

Création d'une page 404 pour une route non existante

Nous n'avons actuellement aucune page 404 pour un itinéraire inexistant. C'est-à-dire, disons que nous tapons **//localhost:4000/sdfsfs**,

Nous obtenons une erreur " Cannot GET / sdfsfs ".

En règle générale, les sites Web affichent une page " 404 not found " chaque fois qu'un utilisateur se rend sur un itinéraire indéfini.

Nous allons créer une page « 404 non trouvée » pour les routes non définies.

Dans le dossier **views**, créez le fichier **notfound.ejs**. Copier le contenu de n'importe quelle page existante EJS comme modèle de départ et remplacez son en-tête h1 par « 404 désolé rien ici ».

Dans mon cas, j'ai copié depuis login.ejs.

Ensuite dans index.js, après enregistrement de toutes les routes à la fin du fichier, ajoutez :

```
app.use((req, res) => res.render('notfound'));
```

Avec cette route de type middleware, Express passera par toutes les routes et s'il ne peut pas en trouver une qui correspond, il affichera la page introuvable.

Essayez maintenant d'entrer un itinéraire non défini comme `//localhost:4000/sdfsfs` et vous devriez voir la page notfound affichée.

Résumé

Nous avons utilisé des sessions avec le middleware de session express pour garder les utilisateurs connectés dans notre application en conservant leurs informations stockées dans le navigateur, puis envoyé ces informations au serveur chaque fois qu'un utilisateur faisait une demande.

Nous avons protégé les pages réservées aux utilisateurs connectés avec un middleware d'authentification et afficher sous condition les liens « Nouvelle publication », « Connexion » et « Nouvel utilisateur » en fonction de la connexion d'un utilisateur ou non.

Nous avons également créé une page 404 pour les itinéraires inexistants.

Chap. 12 Afficher la validation des erreurs

Actuellement, lorsque nous avons des erreurs dans la soumission du formulaire, nous ne les affichons pas à l'utilisateur, mais il suffit de les rediriger vers une autre page.

Ceci est inapproprié car il n'informe pas l'utilisateur qu'il a saisi des données invalides. Ils pourraient penser que ce qu'ils ont entré a été fait avec succès alors qu'en fait, le processus a échoué.

Une meilleure solution serait d'afficher d'emblée les erreurs de validation à l'utilisateur afin qu'il puisse soumettre à nouveau les informations demandées.

Dans ce chapitre, nous verrons comment afficher les erreurs de validation dans les formulaires.

Actuellement dans **storeUser.js**, nous avons le code suivant :

```
const User = require('../models/User.js')
const path = require('path')
module.exports = (req, res) => {
  User.create(req.body, (error, user) => {
    if(error) {
      return res.redirect('/auth/register')
    }
    res.redirect('/')
  })
}
```

L'objet d'erreur nous fournit en fait des notifications d'erreur de validation avec lesquels nous pouvons travailler pour afficher les erreurs.

Affichons à la console les erreurs si nous validons un formulaire d'inscription vide :

Autrement dit, `error` contiendrait une liste d'objets d'erreur de validation individuels avec les clés 'nom d'utilisateur' et 'mot de passe' qui sont les champs respectifs qui ont des erreurs de validation.

Si le nom d'utilisateur est renseigné et est valide, nous n'aurons que l'erreur `key` pour mot de passe.

Nous formatons maintenant les erreurs afin de pouvoir les présenter à l'utilisateur.

Parce que **`Object.keys(error.errors)`** nous donne toutes les clés dans l'objet d'erreurs, nous pouvons le faire avec les éléments suivants pour obtenir les messages d'erreur individuels :

```
const User = require('../models/User.js')
const path = require('path')
module.exports = (req, res) => {
  User.create(req.body, (error, user) => {
    if (error) {
      Object.keys(error.errors).map(key =>
error.errors[key].message)
      return res.redirect('/auth/register')
    }
    res.redirect('/')
  })
}
```

Nous mappons à travers les clés du tableau `error.errors` et pour chacune d'entre elles, accédons au propriété de message d'erreur de la clé. Si vous les connectez à la console, vous obtenez quelque chose comme:

```
[ 'Path `username` is required.',
  'Path `password` is required.' ]
```

Affichage des erreurs de validation dans le modèle

Comment rendons-nous les messages disponibles pour la vue lorsque nous redirigeons en utilisant **return res.redirect('/auth/register')** ?

Nous le faisons en attribuant le message d'erreur dans une variable et en l'enregistrant dans notre session avec les codes ci-dessous en gras :

```
const User = require('../models/User.js')
const path = require('path')
module.exports = (req, res) => {
  User.create(req.body, (error, user) => {
    if (error) {
      const validationErrors = Object.keys(error.errors).map(key =>
error.errors[key].message)
      req.session.validationErrors = validationErrors
      return res.redirect('/auth/register')
    }
    res.redirect('/')
  })
}
```

Et dans **controllers/newUser.js**, on récupère les erreurs de la session avec **req.session.validationErrors** et transmettez-le à **register.ejs** :

```
module.exports = (req, res) => {
  res.render('register', {
    errors: req.session.validationErrors
  })
}
```

Ensuite dans **register.ejs**, nous affichons les messages d'erreur en haut de notre formulaire avec le code ci-dessous placé au-dessus de l'action du formulaire :

```
<div class="container">
<div class="row">
<div class="col-lg-8 col-md-10 mx-auto">
<% if(errors != null && errors.length > 0){ %>
<ul class="list-group"></ul>
<% for (var i = 0; i < errors.length; i++) { %>
<li class="list-group-item list-group-item-danger"><%=
errors[i] %></li>
<% } %>
</ul>
<% } %>
<form action="/users/register" ...>
```

Tout d'abord, nous vérifions que le tableau d'erreurs transmis par `newUser.js` n'est pas nul et a au moins un élément d'erreur.

Si les erreurs sont nulles ou vides, cela signifie qu'il n'y a aucune erreur et nous pouvons continuer avec la soumission du formulaire. S'il y a au moins un élément dans le tableau des erreurs, nous le parcourons et pour chaque erreur, nous affichons avec le `class="list-group-item list-group-item-danger"` pour la faire apparaître en rouge.

Maintenant, lorsque vous exécutez votre application, essayez de soumettre le formulaire « nouvel utilisateur » et si s'il y a des erreurs de validation, il apparaîtra à l'utilisateur en danger rouge.

Erreur d'entrée en double

Cependant, nous rencontrons une erreur critique si nous saisissons un utilisateur non unique, par ex. user1 qui existe déjà dans la base de données, notre application se bloque.

Ceci est dû au fait que nous obtenons une erreur de validation Mongoose lorsque vous tentez de violer une contrainte unique, plutôt qu'une erreur E11000 de MongoDB.

Le package mongoose-unique-validator nous aide à faire une telle gestion des erreurs

Installez d'abord le paquet :

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

puis appliquez-le à /models/User.js avec les codes en gras :

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema;
var uniqueValidator =
  require('mongoose-unique-validator');
const bcrypt = require('bcrypt')
const UserSchema = new Schema({
  username: {
    type: String,
    required: true,
    unique: true},
  password: {
    type: String,
    required: true
  }
});
UserSchema.plugin(uniqueValidator);
```

Maintenant, lorsque nous essayons de sauvegarder un utilisateur, le

validateur unique vérifiera les doublons de la base de données et les signalera comme n'importe quelle autre erreur de validation.

Vider les messages d'erreur de la session

Parce que nous enregistrons les messages d'erreur dans la session et parce qu'une session est permanente, cela signifie que dans le prochain cycle de vie de la demande, la session sera toujours là.

C'est-à-dire que les notifications d'erreur ne sont pas éliminées lorsque l'utilisateur revisite le formulaire après une soumission réussie.

Ainsi, nous avons besoin que les données d'erreur expirent après le cycle de vie actuel de la demande et c'est ce que nous appelons le ringage (flushing), où nous ne voulons pas que les données soient disponible après la prochaine demande.

Nous allons le faire avec un package appelé **connect-flash** (<https://www.npmjs.com/package/connect-flash>). **connect-flash** fournit une zone spéciale de la session utilisée pour stocker les messages. Les messages peuvent être écrits dans cette zone et effacés après avoir été affichés à l'utilisateur.

D'abord installer connect-flash :

```
npm install connect-flash
```

Importez le paquet dans **index.js** :

```
const flash = require('connect-flash');
```

Puis utiliser le middleware avec :

```
app.use(flash())
```

Ensuite, dans **controllers/storeUser.js**, apportez les modifications suivantes :

```
module.exports = (req,res)=>{
  User.create(req.body, (error, user) => {
    if(error){
      const validationErrors =
        Object.keys(error.errors).map(key =>
          error.errors[key].message)
      req.flash('validationErrors',validationErrors)
      //req.session.validationErrors = validationErrors
      return res.redirect('/auth/register')
    }
    res.redirect('/')
  })
}
```

Avec le middleware flash, toutes les requêtes auront une fonction req.flash() qui pourra être utilisée pour les messages flash.

Dans flash(), nous spécifions que les erreurs de validation sera stocké dans la clé 'validationErrors'.

Pour récupérer les erreurs et les présenter dans la vue, effectuez les modifications dans newUser.js :

```
res.render('register',{
  //errors: req.session.registrationErrors
  errors: req.flash('validationErrors')
})
```

C'est-à-dire que nous récupérons les erreurs de la clé 'validationErrors' et les rendons disponibles pour cette vue à afficher, ce après quoi le flash est effacé.

Désormais, lorsque vous exécutez votre application et qu'il y a des erreurs de validation de formulaire, la page des erreurs ne sont là qu'après la première soumission du formulaire.

Les erreurs ne sont plus là lors du rafraîchissement de la page. Ainsi, nous avons pu faire en sorte qu'un utilisateur ne voit que les erreurs pour cette demande spécifique faite.

C'est le rinçage (flushing), où les erreurs de validation ne sont disponibles que pour le cycle de vie suivant la demande et dans le cycle suivant, tout est supprimé.

Personnalisation des messages d'erreur

Actuellement, l'erreur affichée à l'utilisateur n'est pas très utile car elle est trop technique pour qu'ils la comprennent.

Nous devrions fournir des messages d'erreur plus clairs comme « Veuillez fournir un nom d'utilisateur » pour qu'ils rectifient l'erreur.

Pour fournir des messages personnalisés, allez au schéma `models/User.js` et apportez la modification suivante :

```
const UserSchema = new Schema({
  username: {
    type: String,
    required: [true, 'Rentrez SVP un nom d'utilisateur'],
    unique: true
  },
  password: {
    type: String,
    required: [true, 'SVP rentrez un mot de passe']
  }
});
```

C'est-à-dire que **required** accepte un tableau où nous passons le message personnalisé dans le 2ème argument.

Maintenant, nous obtenons le message de validation personnalisé.

Conserver les données de demande sur le formulaire

Actuellement, lorsque l'utilisateur remplit des données dans le formulaire, s'il essaie de les soumettre et il y a une erreur, toutes les données saisies sont supprimées et l'utilisateur doit retaper les données.

Une meilleure expérience utilisateur serait que les données saisies devraient rester et ne pas être supprimées même s'il y a eu des erreurs de validation.

Pour conserver les données saisies, nous flashons également `req.body` qui contient les données insérées dans le formulaire.

Dans **`controllers/storeUser.js`**, ajoutez la ligne suivante :

```
const User = require('../models/User.js')
module.exports = (req,res)=>{
  User.create(req.body, (error, user) => {
    if(error){
      const validationErrors =
        Object.keys(error.errors).map(key =>
          error.errors[key].message)
      req.flash('validationErrors',validationErrors)
      req.flash('data',req.body)
      return res.redirect('/auth/register')
    }
    res.redirect('/')
  })
}
```

Nous stockons req.body dans la clé 'data' de flash. Ensuite dans **controllers/newUser.js**, ajoutez les codes en gras :

```
module.exports = (req, res) =>{  
  var username = ""  
  var password = ""  
  const data = req.flash('data')[0];  
  if(typeof data !== "undefined"){  
    username = data.username  
    password = data.password  
  }  
  res.render('register',{  
    errors: req.flash('validationErrors'),  
    username: username,  
    password: password  
  })  
}
```

Si nous essayons de connecter req.flash(' data ') à la console, nous réalisons que req.flash(' data ') nous renvoie un tableau avec les données dans le premier élément.

Ainsi nous y accédons en utilisant req.flash('data')[0].

Nous vérifions ensuite si req.flash('data') est undefined ce qui sera le cas chaque fois que nous visiterons le nouveau formulaire utilisateur pour la première fois.

Si ce n'est pas indéfini, alors nous attribuons seulement les champs nom d'utilisateur et mot de passe.

Notez que les champs nom d'utilisateur et mot de passe sont initialisés avec une chaîne vide. Ceci afin d'éviter que le formulaire renvoie une erreur indiquant que la valeur est nulle ou indéfinie.

Maintenant que nous avons envoyé les valeurs du nom d'utilisateur et du mot de passe à **register.ejs**, dedans, remplissez les valeurs de données dans la propriété value des balises <input> :

```
...  
<input type="text" class="form-control" placeholder="User  
Name" name="username" value="<%=username %>">  
...  
<input type="password" class="form-control"  
placeholder="Password" name="password" value=" <%=  
password %>">  
...
```

Lorsque vous exécutez l'application maintenant et que l'utilisateur remplit les données dans le formulaire , essayez de submit et il y a une erreur de validation, les données saisies seront conservées et l'utilisateur n'a pas à retaper les données.

Chap. 13 Lier les collections de posts avec les collections de Users

Actuellement, nous n'avons pas lié les utilisateurs avec leurs messages. Nous allons maintenant lier la collection Users and Posts ensemble.

Dans models/BlogPost.js, changez le username en userid avec un objet :

```
const BlogPostSchema = new Schema({
  title: String,
  body: String,
  //username: String,
  userid: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User',
    required: true
  },
  datePosted: {
    type: Date,
    default: new Date()
  },
  image: String
});
```

userid contient un objet avec les propriétés **type**, **ref** et **required**. Dans le type, vous spécifiez **mongoose.Schema.Types.ObjectId** ce qui signifie que la valeur est censée être un identifiant d'objet Mongo valide.

Comme mentionné précédemment, Mongo a un identifiant spécifique pour chaque document et ils doivent être dans un format valide. Dans ref, nous spécifions User pour faire référence à la collection User où le document est contenu.

Ensuite, dans storePost.js, ajoutez la ligne suivante :

```
module.exports = (req,res)=>{
  let image = req.files.image;
  image.mv(path.resolve(__dirname, '..', 'public/img', image.name), async (error)=>{
    await BlogPost.create({
      ...req.body,
      image: '/img/' + image.name,
      userid: req.session.userId
    })
    res.redirect('/')
  })
}
```

C'est-à-dire que nous attribuons l'ID utilisateur avec **req.session.userId**.

Rappelez-vous que **req.session.userId** est renseigné avec l'identifiant de l'utilisateur connecté dans **loginUser.js** lorsqu'un utilisateur se connecte.

Désormais, lorsque vous exécutez votre application, connectez-vous et soumettez un article de blog, le document de publication aura le champ userid rempli.

Cependant, le nom d'utilisateur n'apparaît plus dans la section " Publié par ... " dans l'extrait de billet de blog. C'est parce que nous avons remplacé le nom d'utilisateur par l'identifiant d'utilisateur.

Dans la section suivante, nous verrons comment remplir correctement le message avec ses données d'utilisateur.

Afficher la publication avec les données utilisateur

Notre collection d'articles est maintenant liée à la collection des utilisateurs. Chaque poste stocke une référence à l'utilisateur (userid) qui a créé ce message.

Avec userid, nous allons ensuite remplir le message avec les données de l'utilisateur.

Dans **controllers/home.js**, appelez la méthode `populate('userid')` de `schema.find({})` comme indiqué :

```
const BlogPost = require('../models/BlogPost.js')
module.exports = async (req, res) =>{
  const blogposts = await
  BlogPost.find({}).populate('userid');
  console.log(req.session)res.render('index',{
  blogposts
});
}
```

BlogPost.find({}).populate('userid') référence automatiquement le document avec l'ID utilisateur dans la collection.

Alors maintenant, dans **views/index.ejs**, sous « Publié par », effectuez la modification suivante :

```
...
<p class="post-meta">Posté par
<a href="#"><%= blogposts[i].userid.username %></a>
...
```

Lorsque vous exécutez votre application, vous pouvez obtenir une erreur car certains de vos anciens articles de blog n'ont pas encore de champ `userid` car le code **req.session.userId** n'était pas encore en place.

Dans ce cas, vous devrez peut-être effacer toute la collection de billets de blog dans MongoDB Compass en cliquant sur l'icône de la poubelle.

Une fenêtre contextuelle apparaîtra vous demandant de confirmer la suppression en saisissant le nom de la collection.

Lorsque vous le faites, la collection sera supprimée. Maintenant, soumettez un nouvel article de blog et lorsque vous accédez à la page d'accueil, le nom d'utilisateur sera renseigné.

Nous devons effectuer un changement similaire dans la page de chaque article de blog. Aller à **contrôleurs/getPost.js** :

```
const BlogPost = require('../models/BlogPost.js')
module.exports = async (req,res)=>{
  const blogpost = await
  BlogPost.findById(req.params.id).populate('userid');
  console.log(blogpost)res.render('post',{
  blogpost
  });
}
```

Encore une fois, nous appelons la méthode populate de findById. Dans **views/post.ejs**, remplacez **blogpost.userid.username** pour avoir le nom d'utilisateur apparaître :

```
<span class="meta">Posted by
  <a href="#"><%= blogpost.userid.username %></a>
```

Résumé

Dans ce chapitre, nous avons lié notre collection d'utilisateurs et de messages avec la méthode **schema.find.populate** qui référence le document dans la collection.

Chap. 14 Ajouter un éditeur WYSIWYG

Actuellement, notre formulaire de création d'article de blog n'est qu'un formulaire en texte brut. Nous allons maintenant ajouter un éditeur 'What you see is what you get' - WYSIWIG pour un utilisateur à formater son article de blog exactement dans le style qu'il veut.

Pour ce faire, nous utiliserons un éditeur HTML (<https://summernote.org/>) que nous pouvons joindre à notre blog formulaire de publication

Pour intégrer l'éditeur dans notre formulaire de création de publication, suivez les instructions sur <https://summernote.org/getting-started/#for-bootstrap-4>. Dedans, il y a un exemple avec la façon d'intégrer l'éditeur dans votre page.

Nous devons d'abord copier les lignes suivantes dans le header de notre page :

```
<script
src="https://code.jquery.com/jquery-3.2.1.slim.min.js"></
script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.1
1.0/umd/popper.min.js"></script>
<script
src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta
/js/bootstrap.min.js"></script>
<link
href="https://cdnjs.cloudflare.com/ajax/libs/summernote/0
.8.12/summernote-bs4.css"
```

```
rel="stylesheet">
<script
src="https://cdnjs.cloudflare.com/ajax/libs/summernote/0.
8.12/summernote-bs4.js"></script>
```

Rappelez-vous que notre header est spécifié dans `/views/layout/header.ejs` qui est inclus par toutes les autres pages.

Ainsi, nous devrions ajouter ce qui précède dans **header.ejs**.

Mais si nous le faisons, les scripts seront disponibles pour toutes les pages et ce serait inutile et ajouterait du temps de chargement.

Nous devrions rendre les scripts accessibles uniquement à la page pour créer un article, c'est-à-dire **create.ejs**.

Pour ce faire, dans **controllers/newPost.js**, nous avons un booléen supplémentaire **createPost** qui est défini sur **true** dans **newPost.js**.

```
module.exports = (req, res) =>{
  if(req.session.userId){
    return res.render("create",{
      createPost: true
    });
  }
  res.redirect('/auth/login')
}
```

C'est-à-dire que **createPost** n'existera et n'aura la valeur **true** que lorsqu'un utilisateur visitera l'itinéraire `/posts/new`.

Dans `views/layouts/header.ejs`, nous ajoutons un `if` pour vérifier si `createPost` est défini et est vrai, alors nous incluons les scripts Summernote associés :

```
<!-- Custom styles for this template -->
<link href="/css/clean-blog.min.css" rel="stylesheet">
<% if(locals.createPost && createPost) { %>
<script
src="https://code.jquery.com/jquery-3.2.1.slim.min.js"></
script>
... Les autres scripts de summer note
<% } %>
</head>
```

Notez que l'objet `locals` contient des propriétés qui sont des variables locales dans l'application. Ainsi, pour tester si `createPost` est défini, nous vérifions `locals.createPost`.

Enfin dans `views/create.ejs`, à l'endroit où l'on veut que l'éditeur apparaisse, on déclare un identifiant personnalisé (j'ai choisi `body`) et ajouter le `<script>` comme indiqué au dessous de:

```
<div class="control-group">
<div class="form-group floating-label-form-group
controls">
<label>Description</label>
<textarea id="body" name="body"
class="form-control"></textarea>
<script>
$( '#body' ).summernote({
placeholder: 'Hello bootstrap 4',
tabsize: 2,height: 200
});
</script>
</div>
</div>
```

Et si vous visitez d'autres pages et voyez le code source de la page, les scripts summernote ne seront pas là. Ce n'est que dans la page de création de publication que les scripts summernote seront utilisés. Maintenant, vous pouvez créer des messages avec différents formats

Cependant, une fois le message soumis, le message n'apparaît toujours pas bien formaté. C'est parce que nous n'avons pas demandé au moteur de création de modèles de formater le contenu en HTML au lieu de texte.

Pour dire à EJS de rendre le contenu en HTML au lieu de texte, dans **views/post.ejs**, spécifiez simplement `<%- ... %>` comme indiqué ci-dessous

```
<!-- Post Content -->
<article>
<div class="container">
<div class="row">
<div class="col-lg-8 col-md-10 mx-auto">
<%- blogpost.body %>
</div>
</div>
</article>
```

<https://ejs.co/#docs>

Résumé

Nous avons ajouté un éditeur WYSIWYG pour notre formulaire de création d'article de blog afin de formater l'article de blog.

Chap. 15 MongoDB Atlas

Actuellement, nous allons sur localhost:4000 pour accéder à notre site Web car notre ordinateur agit en tant que serveur pour fournir les informations à notre site Web.

Nous utilisons également un serveur MongoDB en local.

Bien que ce soit idéal pour le développement et le test de nos applications, personne d'autre ne peut actuellement accéder à notre site Web sur Internet.

Dans ce chapitre, nous explorons l'utilisation du service cloud de MongoDB, MongoDB Atlas pour héberger notre base de données sur le cloud et dans le prochain chapitre, nous verrons comment héberger notre serveur Node.js sur Heroku afin que nous puissions livrer notre site web sur internet.

Configuration de MongoDB Atlas

Tout d'abord, inscrivez-vous pour un compte MongoDB Atlas (<https://www.mongodb.com/cloud/atlas>). Sous "Déployer un cluster gratuit", créez un nouveau compte et cliquez sur « Commencer gratuitement ».

Ensuite, dans le panneau de gauche, sous « Sécurité », cliquez sur « Accès à la base de données » où vous n'avez pas encore d'utilisateur.

Créez un utilisateur de base de données en cliquant sur « Ajouter un nouvel utilisateur » et lui fournir les privilèges « Lire et écrire dans n'importe quelle base de données »

Ensuite, sous « Sécurité », « Accès réseau », « Liste blanche IP », sélectionnez « Ajouter une adresse IP » et choisissez « autoriser l'accès depuis n'importe où »

Ensuite, nous nous connecterons avec MongoDB Compass. De retour dans le tableau de bord, sous le cluster que nous venons de créer, cliquez sur 'Connect'

Sélectionnez « Se connecter avec MongoDB Compass ».

Dans le formulaire suivant qui s'affiche, car nous avons déjà Compass, spécifiez votre version et copiez le lien dans un éditeur de texte et remplacez <mot de passe> par le mot de passe de l'utilisateur de base de données nouvellement ajouté.

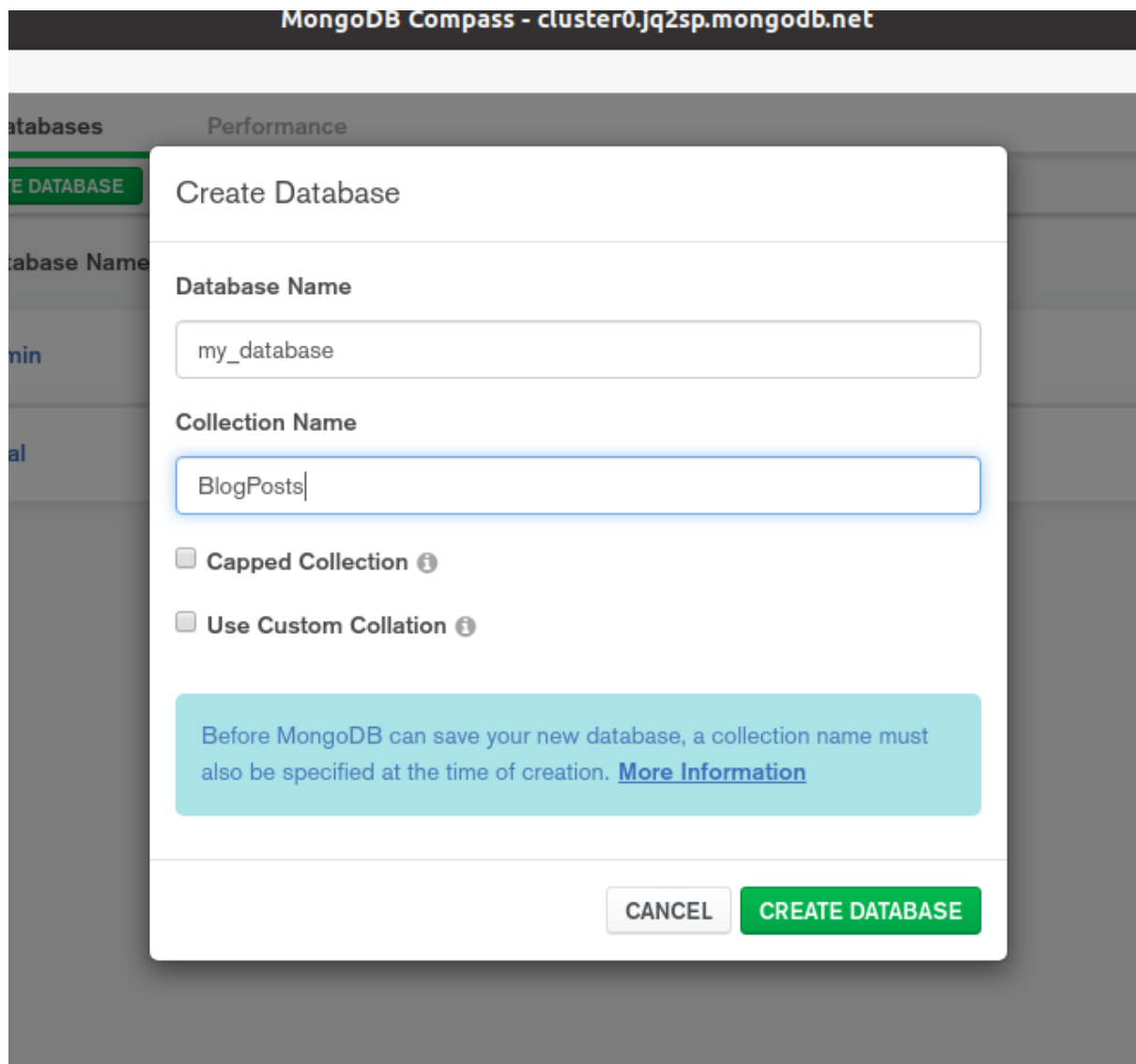
Ensuite, copier à nouveau ce lien.

Ensuite, dans Compass, dans la barre supérieure, sélectionnez « Connecter », « Connecter à ... » et collez le lien.

En fait, Compass peut vous indiquer qu'il a détecté une chaîne de connexion dans le presse-papiers et la remplit automatiquement pour vous.

Cliquez sur « Connecter » et il se connectera ensuite à l'Atlas MongoDB Cluster Cloud.

Une fois connecté, dans MongoDB Compass, cliquez sur « Créer une base de données » et créer une base de données avec la collection « BlogPosts »



Maintenant que nous nous sommes connectés au cluster MongoDB Atlas Cloud de Compass, nous devons ensuite le connecter depuis notre application.

Lors du démarrage de notre application sur une machine locale, nous avons exécuté le serveur MongoDB localement.

Maintenant que nous utilisons le cluster cloud, nous pouvons terminer l'instance de serveur MongoDB mongod sur notre machine.

Et dans index.js, nous devons changer le lien de :

```
mongoose.connect('mongodb://localhost/my_database',  
{useNewUrlParser: true});
```

vers (mettez votre lien) :

```
mongoose.connect('mongodb+srv://newuser1:<your_password>@  
cluster0-vxjpr.mongodb.net/my_database',  
{useNewUrlParser: true});
```

Lancez maintenant votre application et vous vous rendrez compte que tous les utilisateurs et les articles de blog ont disparu car nous nous sommes connectés à une nouvelle base de données hébergée sur le cloud.

Alors, allez-y et ajoutez un nouvel utilisateur, connectez-vous, puis créez un blog des postes. Revenez ensuite à Compass pour voir que l'utilisateur et l'article de blog enregistrés ont été ajoutés à la base de données cloud !

Dans le prochain chapitre, nous allons déployer notre application sur Heroku pour rendre notre application accessible depuis n'importe où sur Internet.

Chap. 16 Déployer une appli web sur Heroku

Heroku nous permet de déployer notre application Node.js sur Internet. Nous allons déployer notre code sur les serveurs d'Heroku qui hébergeront et exécuteront notre Application express qui se connecte à notre base de données cloud MongoDB Atlas.

Le processus de déploiement est relativement simple et vous pouvez simplement suivre les instructions de la documentation pour déployer les applications Node.js sur Heroku (<https://devcenter.heroku.com/>). Mais nous vous guiderons tout au long du processus de déploiement dans ce chapitre.

Tout d'abord, vous aurez besoin d'un compte Heroku. Alors, allez-y et inscrivez-vous pour avoir un compte.

Ensuite, nous devons installer l'interface de ligne de commande Heroku pour créer et gérer nos applications Express sur Heroku.

Une fois l'installation terminée, nous pouvons commencer à utiliser les commandes heroku dans notre Terminal.

Tapez **heroku login** et un navigateur Web s'ouvrira pour la page de connexion Heroku.

Rendre notre application « Heroku Ready »

Avant de commencer à déployer sur Heroku, nous devons faire de notre application « Heroku ready » Nous le faisons dans les sections suivantes en :

- ajout d'un profil
- ajouter notre version Node.js à package.json
- écouter sur le bon port et spécifier le fichier .gitignore

Ajouter un Procfile

Dans notre répertoire d'applications, créez un fichier nommé Procfile (P majuscule, sans extension). Ce fichier sera exécuté lorsque Heroku démarrera notre application. Dans notre simple app, ce fichier ne comportera qu'une seule ligne.

Copiez la ligne ci-dessous dans Procfile :

```
web: node app.js
```

web fait référence au type de processus (le seul type de processus pouvant recevoir du trafic HTTP du Web). La commande après web, c'est-à-dire node app.js, est exécutée sur le Serveur Heroku tout comme ce que nous avons fait pour exécuter notre application sur la machine locale.

package.json

Ensuite, ajoutez la version de Node.js dont votre application a besoin dans package.json. C'est-à-dire, recherchez la version de Node que vous exécutez à l'aide de node --version et ajoutez dans votre package.json comme dans le code ci-dessous en gras :

```
...  
"engines":{  
  "node": "10.19.0"  
},  
"author": "Start Bootstrap",
```

Cela indique à Heroku que notre application nécessite Node 10.19.0 par exemple.

Écouter sur le bon port

Étant donné que Heroku configure une variable d'environnement `process.env.PORT` pour le port, nous devons le spécifier dans `index.js` où nous utilisons `app.listen`.

C'est-à-dire qu'actuellement nous faisons :

```
app.listen(4000, ()=>{  
  console.log('App listening on port 4000 ...')  
})
```

Le remplacer par :

```
let port = process.env.PORT;  
if (port == null || port == "") {  
  port = 4000;  
}  
app.listen(port, ()=>{  
  console.log('App listening...')  
})
```

Cela garantit que notre application sur Heroku écoute sur le port spécifié par `process.env.PORT`.

.gitignore

Ensuite, si nous ne l'avons pas déjà fait, créez un fichier `.gitignore` dans notre application. Ce fichier dit à Git d'ignorer tout ce qui y est spécifié et ne sera pas envoyé sur le serveur.

Et parce que nous n'avons pas besoin d'envoyer les `node_modules`, ajouter **`node_modules`** à `.gitignore`

Avec ces étapes, notre application est maintenant « prête pour Heroku » et nous pouvons continuer à déployer notre application.

Déploiement

Pour le déploiement, si vous ne l'avez pas déjà fait, vous devez avoir la git installé.

Installez git en suivant les instructions dans <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>, puis configurer git pour la première fois (<https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Installer>).

Lorsque git est installé et configuré, configurez un projet git dans le répertoire racine de l'application avec:

```
git init
```

Ensuite, utilisez :

```
git add .
```

pour ajouter tous nos fichiers de projet. Ensuite, pour valider les modifications apportées à votre projet Git, puis :

```
git commit -m 'Initial Commit'
```

```
heroku create
```

Cela crée une nouvelle application vide sur Heroku avec un Dépôt vide Git associé. Une nouvelle URL pour votre application Heroku sera également configurée.

Vous pouvez modifier l'URL ou associer un nom de domaine que vous possédez à l'adresse Heroku mais cela dépasse le cadre de ce cours.

Maintenant, nous poussons notre code vers le référentiel Git distant que nous venons de créer avec:

```
git push heroku master
```

Cela poussera le code vers les serveurs Heroku et configurera notre application avec leurs dépendances. À l'avenir, lorsqu'il y aura des changements de code dans notre app, exécutez à nouveau **git push heroku master** pour le redéployer.

Et si vous allez sur l'URL qui a été générée pour vous, vous verrez votre application sur Internet.