Traduction : <https://zellwk.com/blog/crud-express-mongodb/#convertkit>

Comment construire une application : créé, lire, modifier et supprimer ( CMLS) ?

Premièrement, Express est un framework conçu pour construire des applications web, grâce à NodeJs. Ça simplifie la création du serveur puisque celui-ci est disponible de base dans Node.

Pour les curieux, node permet en fait d’utiliser le javascript comme langage de communication avec le serveur ( et donc pas besoin d’utiliser le php, dans ce cas-ci).

MongoDB est une base de donnée non relationnelle ( no sql) où vous allez stocker les informations de votre site web.

CRUD ( create read update delete) où CMLS ( créé modifier lire suprimer) en français est pour dire au serveur quelles opérations exécuter parmis celles de l’acronyme. Cela se fait au moyen des méthodes suivantes : POST, GET, PUT and Delete.

CREE = POST : fait qq chose

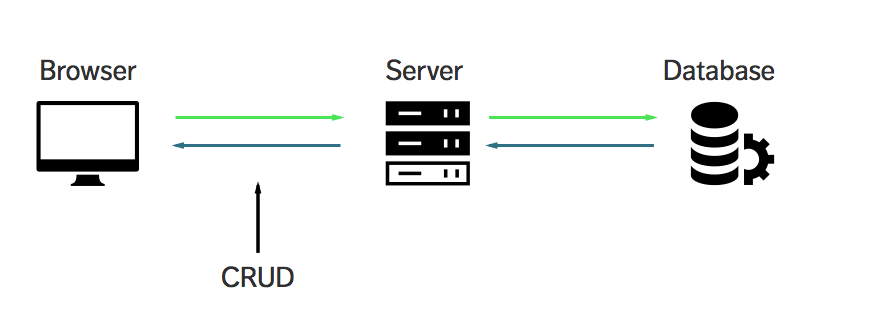
LIRE = GET : prend qq chose

Modifier = PUT : change qq chose

SUPRIMMER = DELETE : enlève qq chose

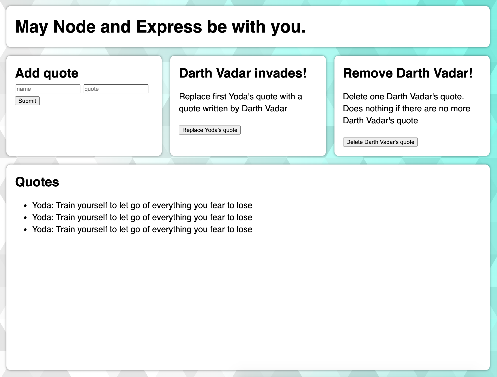
Grâce à ces quatre opérations, il est possible de construire une API REST.

**En résumé, CMLS + Express + Mongo DB =**



**Construisons ensemble une application CMLS basique :**

L’application devrait ressembler à ceci :



***Quels sont les prérequis ?***

1) vérifiez que node est installé en tappant « node -v » dans votre console

2) créer un dossier pour le projet

3) ‘npm init’ dans le dossier du projet

4) complété les champs

5) créer un fichier ‘server.js’

6) console.log dans server.js pour vérifier que Node tourne normalement

7) node server.js dans la console

***Comment utiliser Express :***

1) installez express depuis la console : ‘npm install express –save’

2) importez express dans ‘server.js’ avec ces deux lignes de code :

const express = require('express');

const app = express();

3) créez un serveur auquel les naviguateurs peuvent se connecter, ça se fait avec la méthode ‘listen’ de Express :

app.listen(3000, function() {

console.log('listening on 3000')

})

4) lancez ‘node server.js’ et tappez « localhost :3000 » dans votre naviguateur

5) vous devriez y voir le message suivant : « cannot get / . »

6) c’est bon signe car ça veut dire que maintenant on peut communiquer avec le serveur express via le naviguateur. C’est ici que commence les opérations CMLS.

***CMLS : LIRE***

Sachez qu’un navigateur effectue l’opération « LIRE » quand vous visitez un site web par exemple. En même temps, ils font aussi des requète « GET » au serveur pour effectuer l’opération « LIRE ».

Cannot get / s’affiche car le serveur ne revoit actuellement rien au navigateur.

Avec Express , il faut utiliser la méthode « GET » pour effectuer une requête « GET » :

app.get(endpoint, callback)

Endpoint c’est la valeur qui se place après le nom de domaine.

« callback » indique au serveur quoi faire quand le endpoint requis correspond au endpoint visé.

Voici un exemple ou on remplace le mot fonction par une flèche :

**app.get('/', (req, res) => {**

**res.send('Hello World')**

**})**

La prochaine étape est de modifier « server.js » pour pouvoir afficher une page « index.html » dans le navigateur. Pour faire ça, on utilise la méthode ‘sendFile’ qui est fournie par l’objet ‘res’ (raccourcis de « response » ) :

**app.get('/', (req, res) => {**

**res.sendFile(\_\_dirname + '/index.html')**

// Note: \_\_dirname is the current directory you're in. Try logging it and see what you get!

// Mine was '/Users/zellwk/Projects/demo-repos/crud-express-mongo' for this app.

})

Dans cette méthode, on dit à Express de « servir » la page index.html qui se trouve à la racine de votre projet.

Il faut donc créer ce fichier, qui n’existe pas encore dans le dossier du projet :

Dans la console : touch index.html

Egayons maintenant un peu ce fichier, comme ceci :

**<!DOCTYPE html>**

**<html lang="en">**

**<head>**

**<meta charset="UTF-8"> <title>MY APP</title>**

**</head>**

**<body> <h1> May Node and Express be with you. </h1>**

**</body>**

**</html>**

Il ne vous reste plus qu’a relancer votre serveur et rafraichir le navigateur pour voir le message affiché dans la balise h1.

***Quid de l’outil « Nodemon » ?***

Ça relance le serveur automatiquement, quand vous sauvegardez un fichier utilisé par « server.js ».

Pour l’installer, dans la console, tappez : « npm install nodemon –save-dev »

Pour lancer nodemon server.js sans les trucs chelou de node module, rajoutez ceci dans le « package.json » :

{ // ... "scripts": { "dev": "nodemon server.js" } // ... }

Pour lancer nodemon server.js, exécuter d’abord : « npm run dev ».

***A propos de l’opération : « CREER » :***

Sachez qu’un navigateur peut seulement effectuer une opération de création, à partir du moment où ceux-ci envoient une requête « POST » au serveur. Cette requète peut être effectué avec du JS ou avec un élément « form ».

Comment créer de nouvelle entrée pour notre application ? On va faire ça avec un élément « form » :

Pour envoyer la requète POST, vous devez ajoutez le form à votre index.html, dans lequel vous aurez besoins des trois choses suivantes :

1) un attribut action

2) un attribut méthode

3) un attribut « nom » pour chaques input de l’élément

Ce qui donne :

**<form action="/quotes" method="POST">**

**<input type="text" placeholder="name" name="name">**

**<input type="text" placeholder="quote" name="quote">**

**<button type="submit">Submit</button>**

**</form>**

La méthode indique au naviguateur quelle sorte de requète envoyer, dans ce cas on utilise POST car on envoit une requète POST, c’est assez logique.

L’attribut ACTION dit au naviguateur OU envoyer la requète POST. Dans ce cas, on envoit la requète POST dans « /quotes » ( qui est un sous dossier de notre dossier projet ).

On peut gérer cette requète POST avec une méthode POST dans « server.js »

Le chemin PATH correspond à la valeur que vous avez placé dans l’attribut ACTION

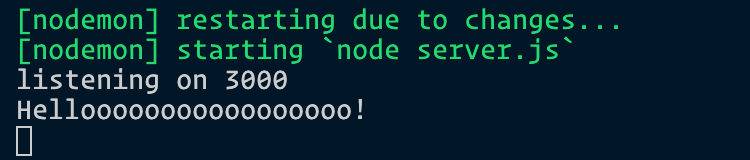
Exemple :

**app.post('/quotes', (req, res) => {**

**console.log('Hellooooooooooooooooo!')**

})

Résultat :



La prochaine étape consiste à se demander : comment obtenir la valeur de l’input avec Express ?

En fait, Express ne gère pas le fait de lire les données du formulaire. Pour qu’il puisse le faire, il faut ajouter un paquet appelé « body-parser ».

Vous le faites via la console : **npm install body-parser –save**

Body-parser est un « middleware ». Ceux-ci nous aide à gérer l’objet « request » avant d’utiliser les middleware concernés.

Pour utiliser des middleware avec Express, on a besoin de la méthode « use » :

Exemple :

**const express = require('express')**

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

**const app = express()**

**// Make sure you place body-parser before your CRUD handlers!**

**app.use(bodyParser.urlencoded({ extended: true }))**

**// All your handlers here...**

**app.get('/', (req, res) => {/\*...\*/})**

**app.post('/quotes', (req, res) => {/\*...\*/})**

La méthode « urlencoded » dans body parser, dit à body parser d’extraire les données du formulaire et de les ajouter à la propriété « body » elle-même contenue dans l’objet « request ».

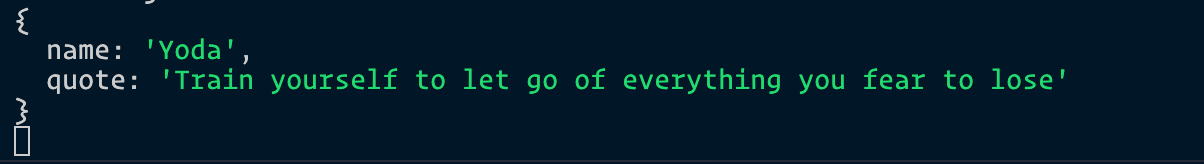
Maintenant, vous devriez être en mesure de voir les valeure du formulaire dans « req.body », exemple :

**app.post('/quotes', (req, res) => {**

**console.log(req.body)**

**})**

Résultat :



***MongoDB :***

On peut stocker des infos dans une base de donnée, pour ensuite les récupérer et les afficher aux personnes qui utiliseront notre application.

Installer MongoDB via la console : **npm install mongodb --save**

Une fois installé, on peut se connecter à MongoDB grâce à la méthode de connection « MongoClient », comme le montre le code ci-dessous :

**const MongoClient = require('mongodb').MongoClient**

**MongoClient.connect('mongodb-connection-string', (err, client) => {**

**// ... do something here**

**})**

La prochaine étape consiste à obtenir le bon lien vers notre base de donnée. Pour ce faire, on va stocker notre base de donnée sur MongoDBAtlas.

1) créer un nouveau projet

2) nommer votre projet

3) construisez un cluster

4) maintenant, il faut connecter notre application au cluster créé.

:

5) dans le cluster, cliquez sur « connect »

6) ajoutez votre adresse ip pour la whitelister

7) créez un utilisateur MongoDB

8) cliquez sur « connectez votre application »

9) prenez votre « connection string » qui devrait ressembler à ceci :

**'mongodb+srv://<username>:<password>@<clustername>-rmp3c.mongodb.net/test?retryWrites=true&w=majority'**

10) connectez vous à la database grâce aux lignes de code suivante :

**MongoClient.connect(connectionString, (err, client) => {**

**if (err) return console.error(err)**

**console.log('Connected to Database')**

**})**

11)vous pouvez utilisez une autre méthode, par exemple avec des promesses plutôt que des callback, grâce à « MongoClient.connect », comme ceci :

**MongoClient.connect(connectionString, { useUnifiedTopology: true })**

**.then(client =>**

**{ console.log('Connected to Database')**

**})**

**.catch(error => console.error(error))**

12) comment changer la base de données ? :

**MongoClient.connect(connectionString, { useUnifiedTopology: true })**

**.then(client => {**

**console.log('Connected to Database')**

**const db = client.db('star-wars-quotes')**

**})**

13) on a maintenant besoin de la variable « db » de la connexion pour accéder à MongoDB, donc on doit placer notre requête express dans l’appel « then » au MongoClient :

**MongoClient.connect(/\* ... \*/)**

**.then(client => {**

**// ...**

**const db = client.db('star-wars-quotes')**

**app.use(/\* ... \*/)**

**app.get(/\* ... \*/)**

**app.post(/\* ... \*/)**

**app.listen(/\* ... \*/)**

**})**

**.catch(console.error)**

***CMLS : CREER***

On doit créer une « collection » avant de pouvoir stocker des items dans la base de données.

Imaginez, c’est comme si la BDD était une pièce et que cette pièce contenait des boites appelés « collection ».

Comme pour les BDD, vous pouvez appelez ces collections comme vous voulez. Dans notre cas, on va stocker des citations dans « quotes » collection. Pour ce faire, on va utiliser **db.collection** pour spécifier la dite collection :

**MongoClient.connect(/\* ... \*/)**

**.then(client => {**

**// ...**

**const db = client.db('star-wars-quotes')**

**const quotesCollection = db.collection('quotes')**

**// ...**

**})**

*Ensuite, on va utiliser la méthode « insertOne » pour ajouter des items à notre collection MongoDB :*

**app.post('/quotes', (req, res) => {**

**quotesCollection.insertOne(req.body)**

**.then(result => {**

**console.log(result)**

**})**

**.catch(error => console.error(error))**

**})**

*Après, pour arrêter le chargement dans le naviguateur ( qui attend une réponse du serveur ) , il faut demander au naviguateur the redirriger vers le dossier racine « / ». On va faire ça grâce à la méthode « res.redirect » :*

***app.post('/quotes', (req, res) => {***

***quotesCollection.insertOne(req.body)***

***.then(result => {***

***res.redirect('/')***

***})***

***.catch(error => console.error(error))***

***})***

***CMLS : operation lire : montrer les citations à l’utilisateur :***

Deux étapes :

1) récupérer les citations à partir de MongoDB Atlas

2) Assurer le rendu des citations en html via un générateur de template.

Pour récupérer les citations dans MongoDB, on va utiliser la méthode « find » :

**app.get('/', (req, res) => {**

**const cursor = db.collection('quotes').find()**

**console.log(cursor)**

**// ...**

**})**

*Cette méthode renvoit un « cursor ». Cet object contient en fait toutes les citations de notre base de données. Il est fourni avec un ensemble de méthode qui nous permettrons de récupérer nos données.*

*On peut par exemple utiliser « toArray » pour convertir nos données dans un tableau :*

**app.get('/', (req, res) => {**

**db.collection('quotes').find().toArray()**

**.then(results => {**

**console.log(results)**

**})**

**.catch(error => console.error(error))**

**// ...**

**})**

La prochaine étape consiste a générer du HTML pour le rendu de nos citations. La première chose à savoir c’est qu’il est impossible d’ajouter dynamiquement du contenu à un fichier html, du moins dans ce cas ci.

Ici, on va utiliser un générateur de template pour générer le HTML. Ici, on va se servir de « embedded javascript : EJS »

1) première étape = installation, via la console : **npm install ejs --save**

2) régler le « view engine » à « ejs ». Cette étape indique à Express qu’on utilise EJS comme générateur de template **: app.set('view engine', 'ejs')** // Middlewares and other routes here...

3) on peut maintenant faire le rendu du HTML

4) on va donc utiliser la méthode « render » construire à l’intérieur de l’objet « response » de Express. Voici la syntaxe appropriée : **res.render(view, locals)**

« view » : c’est le nom du fichier qu’on veut afficher, le fichier doit être placer dans un « view » folder.

« locals » : ce sont les données qu’on envoit dans le fichier

Pour créer la vue, il faut créer un fichier « index.ejs » à l’intérieur de notre dossier « view » :

**mkdir views**

**touch views/index.ejs**

5) copier ensuite tout de « index.html » à « index.ejs » :

**<!-- index.ejs -->**

**<!DOCTYPE html>**

**<html lang="en">**

**<head>**

**<meta charset="UTF-8" />**

**<title>Star Wars Quote App</title>**

**</head>**

**<body>**

**<h1>May Node and Express be with you.</h1>**

**<form action="/quotes" method="POST">**

**<input type="text" placeholder="name" name="name" />**

**<input type="text" placeholder="quote" name="quote" />**

**<button type="submit">Submit</button>**

**</form>**

**</body>**

**</html>**

6) en suite, on utilise « res.render » pour render le index.ejs :

**app.get('/', (req, res) => {**

**db.collection('quotes').find().toArray()**

**.then(/\* ... \*/)**

**.catch(/\* ... \*/)**

**res.render('index.ejs', {})**

**})**

7) pour placer les citations dans « index.ejs », on va envoyer les citations dans la méthode « render » :

**app.get('/', (req, res) => {**

**db.collection('quotes').find().toArray()**

**.then(results => {**

**res.render('index.ejs', { quotes: results })**

**})**

**.catch(/\* ... \*/)**

**})**

8) dans « index.ejs » on peut placer les citations dans le HTML :

**<!-- In index.ejs -->**

**<body>**

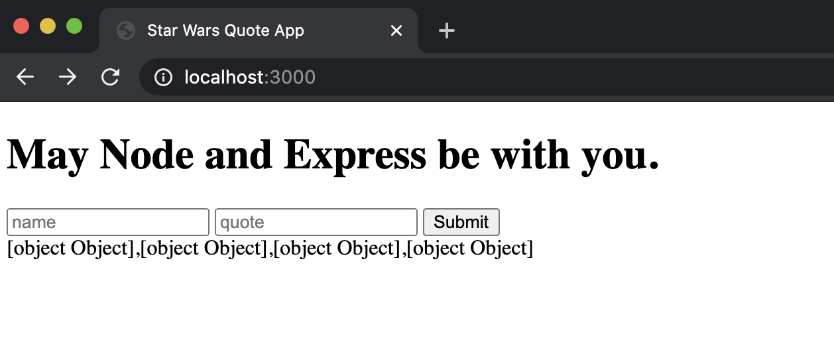
**<h1> ... </h1>**

**<form> ... </form>**

**<%= quotes %>**

**</body>**

Ce qui donne comme résultat :



Object, object partout c’est car chaque citation à l’intérieur de « results » est un object javascript. « ejs » ne peut pas convertir automatiquement cet objet en HTML.

Pour se faire, il faut boucler à travers les citations, on va faire ça avec une boucle « for ».

En ejs, on écrit ça presque comme en JS, la seule différence c’est ça : <% %>

**<h2> Quotes </h2**

**<ul class="quotes">**

**<!-- Loop through quotes -->**

**<% for(var i = 0; i < quotes.length; i++) {%>**

**<li class="quote">**

**<!-- Output name from the iterated quote object -->**

**<span><%= quotes[i].name %></span>:**

**<!-- Output quote from the iterated quote object -->**

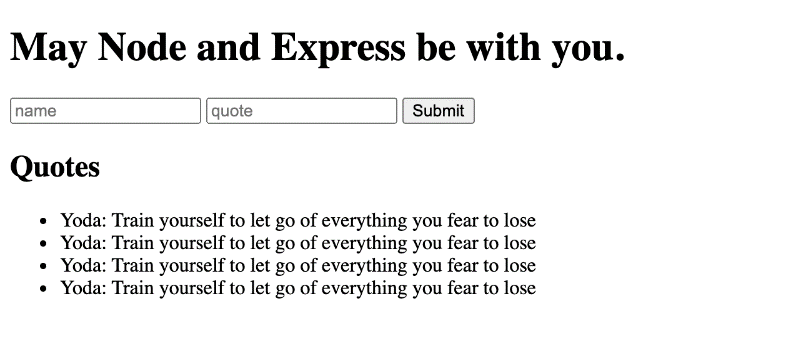
**<span><%= quotes[i].quote %></span>**

**</li>**

**<% } %>**

**</ul>**

Ce qui donne comme résultat :



***CMLS : Modifier***

On utiliser pour modifier qq chose dans la BDD. Pour ce faire on utilise une requète **PUT .**

Cette requête peut être lancé avec du JS ou depuis un formulaire.

Pour l’opération modifier, on va créer un boutton qui remplacera la première citation par Yoda par qq chose écrit par Darth Vadar :

**<div>**

**<h2>Darth Vadar invades!</h2>**

**<p>**

**Replace first Yoda's quote with a quote written by Darth Vadar**

**</p>**

**<button id="update-button">Replace Yoda's quote</button>**

**</div>**

On va aussi créer un fichier JS externe pour exécuter la requète  **PUT.**

**$ mkdir public**

**$ touch public/main.js**

*Ensuite, il faut dire à Express de créer un dossier public accessible au publique, pour ça on utilise un middleware appelé : « express.static » :*

**app.use(express.static('public'))**

*Après, on va ajouter le « main.js » au « index.ejs » file :*

**<body>**

**<!-- ... -->**

**<script src="/main.js"></script>**

**</body>**

*Troisièmement, on va envoyer une* ***PUT*** *requète au moment ou l’utilisateur va cliquer sur le boutton.*

*Ca veut dire qu’on doit « écouter » un évènement « click ».*

**// main.js**

**const update = document.querySelector('#update-button')**

**update.addEventListener('click', \_ => {**

**// Send PUT Request here**

**})**

*La manière la plus simple de déclencher une requète PUT est d’utiliser l’API : « fetch ».*

**fetch(endpoint, options)**

*Dans notre cas, on veut envoyer la requète dans le dossier /quotes donc on règle le « endpoint » à /quotes :*

**update.addEventListener('click', \_ => {**

**fetch('/quotes', {/\* ... \*/})**

**})**

*Maintenant, on a besoin d’envoyer une requète PUT, on peut le faire en réglant la méthode Fetch sur « put » :*

**update.addEventListener('click', \_ => {**

**fetch('/quotes', {**

**method: 'put'**

**})**

**})**

*Les apps modernes renvoit des données au format JSON aux serveurs. Ces apps recoivent également du JSON en retour du serveur.*

*La prochaine étape est donc de dire au serveur qu’on va lui envoyer des datas en JSON en réglant le « content-type » headers sur « application/json » :*

**update.addEventListener('click', \_ => {**

**fetch('/quotes', {**

**method: 'put',**

**headers: { 'Content-Type': 'application/json' },**

**})**

**})**

*Ensuite, on va devoir convertir les données qu’on envoi en JSON. On peut le faire avec « JSON.stringify », la donnée sera passé via la propriété « body » :*

**update.addEventListener('click', \_ => {**

**fetch('/quotes', {**

**method: 'put',**

**headers: { 'Content-Type': 'application/json' },**

**body: JSON.stringify({**

**name: 'Darth Vadar',**

**quote: 'I find your lack of faith disturbing.'**

**})**

**})**

**})**

*Accepter la requète PUT : il faut savoir que le serveur n’accepte pas encore le format JSON car on ne l’a pas configuré pour. On peut lui apprendre à lire du JSON grâce au « body-parser » json middleware :*

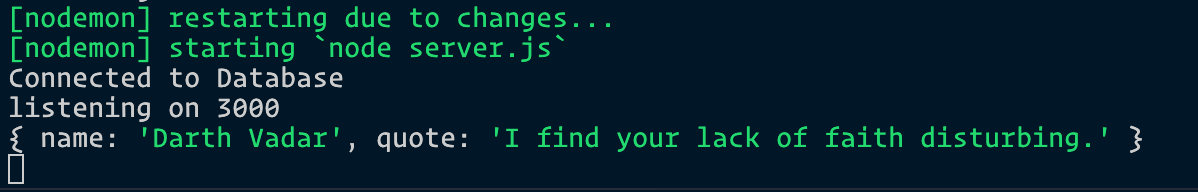
**app.use(bodyParser.json())**

*Ensuite, on peut gérer la requète PUT grâce à la méthode PUT :*

**app.put('/quotes', (req, res) => {**

**console.log(req.body)**

**})**

****

La prochaine étape sera the changer la citation de Yoda par celle de Vadar.

Dans MongoDB Collections, vous trouverez la méthode « findOneAndUpdate », grâce à elle on peut trouver et changer un item dans la BDD, avec la syntaxe suivante :

**quotesCollection.findOneAndUpdate(**

**query,**

**update,**

**options**

**)**

**.then(result => {/\* ... \*/})**

**.catch(error => console.error(error))**

*a) query : sert a filtrer la collection avec des pairs clef-valeurs. Par exemple, si on veut filtrer les citations et les limiter à celles écrites par Yoda :*

**quotesCollection.findOneAndUpdate(**

**{ name: 'Yoda' },**

**update,**

**options**

**)**

**.then(result => {/\* ... \*/})**

**.catch(error => console.error(error))**

*b) update : indique à MongoDB quoi changer, pour ça, on utilise les opérateurs de modification de Mongo tel que : $set ; $inc et $push.*

*Ici on va utiliser l’opérateur $set puisqu’on va changer la citation de Yoda en citation de Darth Vadar :*

**quotesCollection.findOneAndUpdate(**

**{ name: 'Yoda' },**

**{**

**$set: {**

**name: req.body.name,**

**quote: req.body.quote**

**}**

**},**

**options**

**)**

**.then(result => {/\* ... \*/})**

**.catch(error => console.error(error))**

*Dans notre cas, il est possible qu’aucune citation de Yoda n’existe dans la BDD. On peut donc forcer Mongo à créer une nouvelle citation de Vadar si aucune de Yoda n’existe. On va le faire en réglant « upsert » sur « true ».*

*« upsert » veut dire : inserer un document si aucun document ne peut être modifier*

**quotesCollection.findOneAndUpdate(**

**{ name: 'Yoda' },**

**{**

**$set: {**

**name: req.body.name,**

**quote: req.body.quote**

**}**

**},**

**{**

**upsert: true**

**}**

**)**

**.then(result => {/\* ... \*/})**

**.catch(error => console.error(error))**

*Enfin, on va loger le « result » dans la ligne de commande :*

**app.put('/quotes', (req, res) => {**

**quotesCollection.findOneAndUpdate(/\* ... \*/)**

**.then(result => {**

**console.log(result)**

**})**

**.catch(error => console.error(error))**

**}**

*Aussi, il est nécessaire de répondre au JS qui envoit la requète PUT : dans ce cas ci, on va simplement renvoyer un message « success » :*

**app.put('/quotes', (req, res) => {**

**quotesCollection.findOneAndUpdate(/\* ... \*/)**

**.then(result => {**

**res.json('Success')**

**})**

**.catch(error => console.error(error))**

**}**

*Après, on gérer la réponse du serveur avec un objet « then » ( fetch retourne une promesse donc on a pas le choix ). Cependant, Fetch est un peu différent des autres types de promesses. Vous devez donc utiliser un autre objet « then » pour récupérer la réponse du serveur :*

**fetch({ /\* request \*/ })**

**.then(res => {**

**if (res.ok) return res.json()**

**})**

**.then(response => {**

**console.log(response)**

**})**

*On pourrait updater le DOM mais dans ce cas ci on va juste rafraichir le naviguateur pour checker les changements :*

**fetch({ /\* request \*/ })**

**.then(res => {**

**if (res.ok) return res.json()**

**})**

**.then(response => {**

**window.location.reload(true)**

**})**

***CMLS : Suprimmer***

On peut lancer ce type d’opération grâce à une requète « delete ». C’est assez semblable à la requète « modifier ».

Dans notre exemple, on va suprimer la première citation de Darth Vadar :

*Pour ça, on doit d’abord ajouter un boutton « delete » à « index.ejs » :*

**const deleteButton = document.querySelector('#delete-button')**

**deleteButton.addEventListener('click', \_ => {**

**fetch('/quotes', {**

**method: 'delete',**

**})**

**})**

*Puisqu’on supprime une citation de Darth Vadar, on a seulement besoin d’envoyer le nom Darth Vadar au serveur :*

**deleteButton.addEventListener('click', \_ => {**

**fetch(/\* ... \*/, {**

**method: 'delete',**

**headers: { 'Content-Type': 'application/json' },**

**body: JSON.stringify({**

**name: 'Darth Vadar'**

**})**

**})**

**.then(res => {**

**if (res.ok) return res.json()**

**})**

**.then(data => {**

**window.location.reload()**

**})**

**})**

*On va gérer l’évènement sur notre serveur grâce à la requète « delete » :*

**app.delete('/quotes', (req, res) => {**

**// Handle delete event here**

**})**

*Comment suprimer un document de MongoDB : grâce à la méathode « deleteOne » qui va nous permettre de retirer un document de la BDD. Cette méthode a deux paramètres : « query » et « options » :*

**quotesCollection.remove(**

**query,**

**options**

**)**

**.then(result => {/\* ... \*/})**

**.catch(error => console.error(error))**

*QUERY fonctionne comme QUERY dans findOneAndUpdate. Ca nous permet de filtrer la collection aux entrées de la BDD que nous cherchons. Dans notre cas on va « set » le « name » sur DarthVadar :*

**quotesCollection.remove(**

**{ name: 'Darth Vadar' },**

**options**

**)**

**.then(result => {/\* ... \*/})**

**.catch(error => console.error(error))**

*Cependant, puisqu’on a déjà passé le nom DarthVadar à « fetch », on a pas besoin de le recoder dans Express. On peut simplement utiliser : « req.body.name » :*

**app.delete('/quotes', (req, res) => {**

**quotesCollection.remove(**

**{ name: req.body.name },**

**options**

**)**

**})**

*Dans notre cas, on a pas besoin de changer plus d’options donc on peut oublier « options » :*

**app.delete('/quotes', (req, res) => {**

**quotesCollection.deleteOne(**

**{ name: req.body.name }**

**)**

**})**

*Après, on peut envoyer une réponse au javascript dans l’appel « then » :*

**app.delete('/quotes', (req, res) => {**

**quotesCollection.deleteOne(**

**{ name: req.body.name }**

**)**

**.then(result => {**

**res.json(`Deleted Darth Vadar's quote`)**

**})**

**.catch(error => console.error(error))**

**})**

*Maintenant, quand vous cliquerez sur le boutton « delete », le naviguateur enverra une requète « delete » en passant par « fetch » à notre serveur Express. Ensuite, le serveur répondra en envoyant soit une erreur soit un message.*

*Quid s’il n’y a plus de citations de Darth Vadar ? : Si il n’y en a plus, « result.deletedCount » will be « o ». On peut donc envoyer un message qui dira : dit au naviguateur qu’il n’y a plus de citation de Darth Vadar à suprimmer :*

**app.delete('/quotes', (req, res) => {**

**quotesCollection.deleteOne(/\* ... \*/)**

**.then(result => {**

**if (result.deletedCount === 0) {**

**return res.json('No quote to delete')**

**}**

**res.json(`Deleted Darth Vadar's quote`)**

**})**

**.catch(error => console.error(error))**

**})**

*Si le javascript reçoit un « no quote to delete » comme réponse, on peut dire à l’utilisateur qu’il n’y a plus de citation à supprimer, pour ce faire, on va ajouter un élément où on dirait aux utilisateur quid de tout ceci :*

**<div id="message"></div>**

*Si on reçoit un « no quote to delete », on peut changer le « textContent » du « .message » div :*

const messageDiv = document.querySelector('#message')

**deleteButton.addEventListener('click', \_ => {**

**fetch(/\* ... \*/)**

**.then(/\* ... \*/)**

**.then(response => {**

**if (response === 'No quote to delete') {**

**messageDiv.textContent = 'No Darth Vadar quote to delete'**

**} else {**

**window.location.reload(true)**

**}**

**})**

**.catch(/\* ... \*/)**

**})**