Aide sur MongoDB

Pierre Pompidor

Table des matières

1 Le SGBD NoSQL MongoDB **5**

1.1 Pourquoi utiliser une base de données NoSQL 5

1.2 Présentation de MongoDB 6

1.2.1 Les collections et les documents 6

1.2.2 Les index 7

1.3 Mise en œuvre de MongoDB 7

1.3.1 Installation de MongoDB 7

1.3.1.1 Installation de MongoDB sous Linux 7

1.3.1.2 Installation de MongoDB sous Windows ou sous MacOS 8

1.3.1.3 Utilisation de MongoDB en lignes de commandes 8

1.3.2 Liste des bases de données 9

1.3.3 Création d'une base de données 9

1.3.4 Liste des collections 9

1.3.5 Création d'une collection 9

1.3.5.1 Insertion des documents dans une collection . 9

1.3.5.2 Importation de documents à partir d'un fichier 10

1.3.5.3 Exportation des documents d'une collection dans un fichier JSON 10

1.3.6 Interrogation d'une collection 11

1.3.6.1 Principe de l'interrogation via un objet "filtre" 11

1.3.6.2 Les opérateurs de comparaisons, les opérateurs ensemblistes et logiques 12

1.3.6.3 L'opérateur $exists 13

1.3.6.4 L'opérateur *$in* 13

1.3.6.5 L'opérateur *$nin* 13

1.3.6.6 L'opérateur *$or* 13

1.3.6.7 L'opérateur *$not* 13

1.3.6.8 L'opérateur $nor 14
1.3.7 L'application d'expressions régulières 14
1.3.8 Les projections et la méthode *distinct()* 14
1.3.8.1 Les projections 14
1.3.8.2 La méthode *distinct()* 15
1.3.9 Référencement des documents et jointures 15
1.3.9.1 Les objets imbriqués (*nested objects*) 16
1.3.9.2 Les objets référencés 17
1.3.9.3 Les jointures 17
1.3.10 Mise à jour et suppression d'un document 21
1.3.10.1 Mise à jour d'un document 21
1.3.10.2 Suppression d'un document 22
1.3.11 Suppression d'une collection 22
1.4 Utilisation de MongoDB via Node.js 22
1.4.1 Installation du module MongoDB pour Node.js 22
1.4.2 Connexion au serveur MongoDB 23
1.4.3 Insertion de données à partir d'un serveur Node.js . . 24
1.4.4 Interrogation de données à partir d'un serveur Node.js 25
1.4.4.1 Exploitation du résultat de la méthode *find()* 25
1.4.4.2 Utilisation de la méthode toArray() 26
1.4.5 La synchronisation des requêtes 28
1.4.5.1 L'utilisation des fonctions de callback 29
1.4.5.2 L'utilisation du module *async* 30
1.4.5.3 La méthode *async.series()* 32
1.4.5.4 La méthode *async.waterfall()* 33
1.5 Interrogation de MongoDB via les routes gérées par *express* . 34
1.5.1 Structure d'un serveur Node.js interrogeant MongoDB 34
1.5.2 La problématique du Cross-Origin Resource Sharing . 35
1.5.3 Exemples de gestion de routes 35
1.5.3.1 Gestion d'une route pour lister les marques . 36
1.5.3.2 Gestion d'une route pour filtrer les produits . 36
1.5.3.3 Recherche d'un produit à partir de son identifiant interne 37

Chapitre 1

Le SGBD NoSQL MongoDB

MongoDB est un système de gestion de bases de données créé en 2007 par la société homonyme. Ce SGBD fait partie de la famille **NoSQL** (*Not only SQL*) qui regroupe des SGBD non relationnels de différents types. La principale motivation à s'écarter du modèle relationnel est la volonté de s'affranchir des nombreuses contraintes imposées par celui-ci, contraintes qui peuvent devenir gênantes si un accès très rapide à de grandes volumétries de données est privilégié. Cette efficacité est en partie due à ce que les contraintes d'intégrité et les transactions ACID ne sont pas assurées par MongoDB (elle sera aussi due à une création performante et dynamique d'index).

MongoDB stocke les données sous un format binaire appelé **BSON** permettant (à quelques exceptions près) de sérialiser des objets JavaScript en binaire (suivant un format dit clefs/valeurs, les objets JavaScript sont eux-mêmes implémentés par des tableaux associatifs). Le mot BSON s'inspire directement de JSON (*JavaScript Object Notation*) qui est le format textuel de la sérialisation des objets JavaScript.

Parmi la famille des SGBD NoSQL, MongoDB est classé dans la sous-famille des SGBD orientés **documents** (notamment avec le SGBD **Redis**). Schématiquement, un document est donc la représentation d'un objet JavaScript dans une collection (une liste d'objets).

1.1 Pourquoi utiliser une base de données NoSQL

MongoDB a comme principal avantage la capacité de pouvoir gérer une volumétrie très importante de données car une base de données peut être

répartie sur plusieurs serveurs tout en assurant des accès efficaces (le SGBD *Redis* en gardant les données en mémoire vive assure des accès encore plus efficaces mais au dépend d'une certaine vulnérabilité). Cela-dit, ce point n'est pas forcément celui qui nous intéresse principalement dans le cas de notre fil rouge (la création d'une application de commerce en ligne), même si une entreprise de e-commerce peut à terme devoir gérer beaucoup de données entre les dizaines de milliers de références des produits en vente et la base clientèle.
Par ailleurs, MongoDB s'avère un bon choix dans la gestion de données hétérogènes et quand il est difficile de pré-déterminer les attributs des entités qui seraient à l'origine de nos tables ou de nos collections (imaginons par exemple que si nous voulions modéliser le contenu des articles d'une encyclopédie en ligne, il serait bien difficile de prélister tous leurs attributs, et si cette liste était faite, il est certain que la majorité de ces attributs n'auraient pas de valeurs dans de nombreux articles).
Pour revenir à notre étude de cas, notre principale motivation est plutôt l'homogéinité de ce choix dans le cadre d'un développement guidé par JavaScript et la flexibilité dans la représentation des données que nous offre ce langage. Par ailleurs une application de commerce en ligne mobilise relativement peu d'entités, et il assez facile de prendre en charge les contraintes d'intégrité au niveau des traitements.

1.2 Présentation de MongoDB

1.2.1 Les collections et les documents

Si vous êtes habitué à un SGBD relationnel, une première approximation (très grossière car nous ne sommes pas dans un modèle relationnel), est de considérer que les tables vont correspondre aux collections et les tuples aux documents. Cela (mal) dit, il faut tout de suite prendre conscience de ce qui suit :

— sous MongoDB, une base de données n'est pas censée être construite suivant un schéma prédéfini (même s'il sera possible de le faire par exemple avec l'*ODM Mongoose*), et donc **les tuples ne sont pas censés posséder une liste identique d'attributs** ;
— par ailleurs MongoDB n'assurant pas la vérification des contraintes d'intégrité, celle-ci devra être effectuée au niveau de la logique métier.

Un point important à bien comprendre est que MongoDB ne permet qu'une gestion partielle des clefs étrangères. Si vous voulez identifier un do-

cument (que nous appellerons le document cible) d'une collection dans un document (que nous appellerons le document source) d'une autre collection, vous aurez deux solutions :

— la duplication des propriétés du document cible dans le document source ce qui peut paraître maladroit mais facilitera les temps d'accès ;
— l'identification du document cible dans le document source par l'identifiant unique attribué par MongoDB (en fait comme si nous manipulions une clef étrangère) mais cela vous obligera à produire un peu de code.

Ce point est détaillé dans la section "Référencement des documents et jointures".

1.2.2 Les index

Une des grandes forces de MongoDB est la gestion des **index**.
Chaque collection est indexée par défaut sur l'attribut *_id*, mais vous pouvez à tout moment, créer un autre index sur un ou plusieurs autres attributs (en créant un index composé).

Pour voir les index posés sur une collection, vous devez appliquer la méthode **getIndexes()** sur la collection :

```
db.<NomDeLaCollection>.getIndexes()
```

Pour créer un index sur une collection, vous devez utiliser la méthode **createIndex()** :

```
db.<NomDeLaCollection>.createIndex(<attributs>, <options>)
```

Vu la complexité de la création des index et les stratégies qu'elle implique, ce point ne sera pas détaillé dans cet ouvrage car il mériterait un chapitre entier.

1.3 Mise en œuvre de MongoDB

1.3.1 Installation de MongoDB

1.3.1.1 Installation de MongoDB sous Linux

Toutes les informations utilses vous seront données sur ce site :
https ://docs.mongodb.com/manual/installation/

et plus particulièrement pour Ubuntu :
https ://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/

Le lancement du service est effectué par la commande suivante (attention *Systemd* est le gestionnaire de services depuis Ubuntu 16.04)

```
sudo systemctl enable mongodb
```

Et pour vérifier quel est le port occupé par MongoDB (il tourne par défaut sur le port 27017) :

```
sudo netstat -antup
```

1.3.1.2 Installation de MongoDB sous Windows ou sous MacOS

Toutes les informations utilses vous seront données sur ce site :
https ://docs.mongodb.com/manual/installation/

1.3.1.3 Utilisation de MongoDB en lignes de commandes

En lignes de commandes, la gestion des bases de données de MongoDB et des collections contenues dans celles-ci, s'effectue grâce au **mongo shell** qui est une interface JavaScript interactive. Pour cela il suffit d'exécuter l'exécutable **mongo** dans votre terminal ou votre invite de commandes.

```
mongo
```

Il est intéressant de remarquer que le mongo shell peut évaluer (presque) n'importe quel code JavaScript et afficher des résultats via sa méthode **print()**.

```
var me="Pierre"
print("Bonjour", me)
```

La fonction *print()* permet d'afficher des messages sur le terminal ou l'invite de commandes.

Remarque :
Dans les exemples qui suivent, les mots encadrés par < et > sont à remplacer par vos propres données :

- `<NomDeLaBase>` : nom de votre base MongoDB
- `<NomDeLaCollection>` : nom de votre collection MongoDB
- `<Document>` : un document MongoDB càd un objet stocké dans une collection
- `<Collection>` : une collection càd une liste d'objets → `[Objet1, ..., Objetn]`

— `<ObjetFiltre>` : un objet permettant de filtrer une collection MongoDB pour sélectionner les documents correspondants aux critères de recherche
— `<FichierJSON>` : un fichier au format JSON

1.3.2 Liste des bases de données

La liste des bases de données est obtenue par l'instruction suivante :

```
show dbs
```

1.3.3 Création d'une base de données

La création d'une base de données est obtenue par l'instruction suivante :

```
use <nom de la base>
```

mais attention, cette création ne sera effective que si une collection est créée dans la base.

1.3.4 Liste des collections

La liste des collections est obtenue par une des instructions suivantes :

```
show collections
show tables
```

1.3.5 Création d'une collection

Pour gérer une collection dans le *mongo shell*, nous allons appliquer des méthodes à l'objet **db** qui représente la base de données courante.
Pour mettre en œuvre les étapes de création d'une collection, nous créons une base de donnée nommée *OnlineSales* en la sélectionnant (même si elle n'existe donc pas encore) :

```
use OnlineSales
```

Cette base de données sera définitivement créée lors de la création de sa première collection.

1.3.5.1 Insertion des documents dans une collection

La **collection** sera créée lors de l'insertion des premiers **documents** dans celle-ci. La collection peut être (epproximativement) considérée comme étant une table, et un document comme un tuple de celle-ci.

Nous pouvons insérer un seul document ou une collection de documents :

```
db.<NomDeLaCollection>.insert(<Document>)
db.<NomDeLaCollection>.insert(<Collection>)
```

Exemples :

```
db.Products.insert({"type": "phone", "brand": "Peach", "name": "topPhone"})

db.Products.insert([
  {"type": "phone", "brand": "Peach", "name": "topPhone 8 64G"},
  {"type": "phone", "brand": "Peach", "name": "topPhone 8 256G"},
  {"type": "phone", "brand": "Threestars", "name": "bigPhone"},
  {"type": "phone", "brand": "Konia", "name": "Konia 4000"},
  {"type": "headset", "brand": "Earlid", "name": "Earlid pro"}
])
```

Remarque : l'insertion d'une date se fait via la méthode **Date()** du mongo shell.

1.3.5.2 Importation de documents à partir d'un fichier

Les documents peuvent aussi être insérés à partir d'un fichier JSON grâce à la commande *mongoimport*.
Par défaut, les documents sont rajoutés aux documents existants : si vous voulez supprimer les documents existants, utilisez l'option *–drop*.

Attention, il doit y avoir **un objet par ligne** (et les objets ne sont donc pas séparés par des virgules.

```
mongoimport --db <NomDeLaBase> --collection <NomDeLaCollection>
            --file <FichierJSON>
```

Pour insérer une collection (c'est à dire des objets listés dans une liste et donc séparés par des virgules), paramétrez l'instruction précédente avec l'option *–jsonArray* :

```
mongoimport --db <NomDeLaBase> --collection <NomDeLaCollection>
            --file <FichierJSON> --jsonArray
```

1.3.5.3 Exportation des documents d'une collection dans un fichier JSON

Les documents d'une collection peuvent être exportés (par exemple dans le but de faire une sauvagarde) dans un fichier JSON grâce à la commande *mongoexport*.

```
mongoexport --db <NomDeLaBase> --collection <NomDeLaCollection>
            --out <FichierJSON>
```

1.3.6 Interrogation d'une collection

1.3.6.1 Principe de l'interrogation via un objet "filtre"

Pour sélectionner tous les documents de la collection, utilisez la méthode *find()* sans paramètre :

```
db.<NomDeLaCollection>.find()
```

Par exemple :

```
db.Products.find()
```

Il est à remarquer que la méthode *find()* renvoyant une liste d'objets (les documents sélectionnés), vous pouvez accéder au document souhaité via son indice :

```
db.Products.find()[0]
```

renverra donc le premier document de la collection *Products* (si il existe).

et donc à une propriété de ce document, par exemple :

```
db.Products.find()[0].name
```

Pour compter tous les documents de la collection, utilisez la méthode *count()* sur le résultat renvoyé par *find()* :

```
db.<NomDeLaCollection>.find().count()
```

Par exemple :

```
db.Products.find().count()
```

Pour afficher la valeur de certaines propriétés des documents de la collection, scriptez du JavaScript en utilisant la méthode *forEach()* sur le résultat renvoyé par *find()* :

```
db.<NomDeLaCollection>.find().forEach(function(doc){ <codeJavaScript> })
```

Par exemple :

```
db.Products.find().forEach(function(doc){ print(doc.name); })
```

La méthode **find()** peut comporter en paramètre un objet qui va spécifier les critères de sélection des objets d'une collection.
Nous appellerons cet objet "l'objet filtre".

```
db.<NomDeLaCollection>.find(<ObjetFiltre>)
```

Par exemple si nous voulons sélectionner tous les produits de marque *Peach* de notre collection *Products*, notre recherche s'écrira ainsi :

```
db.Products.find({"brand": "Peach"})
```

Il est bien sûr possible dans l'*objet filtre* de spécifier plusieurs critères de recherche :

```
db.Products.find({"brand": "Peach", "type" : "phone"})
```

Il est souvent utile d'utiliser la méthode *toArray()* appliquée à la méthode *find()* qui renvoie une collection.
Par exemple :

```
db.Products.find({"brand": "Peach"}).toArray(function(err, documents) { ... })
```

Pour ne sélectionner que le premier document satisfaisant les critères de recherche, vous pouvez utiliser la méthode **findOne()**.

```
db.<NomDeLaCollection>.findOne(<ObjetFiltre>)
```

1.3.6.2 Les opérateurs de comparaisons, les opérateurs ensemblistes et logiques

Les opérateurs de comparaisons suivants peuvent être utilisés dans *l'objet filtre* :

- **$eq** : renvoie vrai si la valeur de la propriété est égale à la valeur spécifiée
- **$ne** : renvoie vrai si la valeur de la propriété est différente de la valeur spécifiée
- **$lt** : renvoie vrai si la valeur de la propriété est plus petite que la valeur spécifiée
- **$lte** : renvoie vrai si la valeur de la propriété est plus petite ou égale que/à la valeur spécifiée
- **$gt** : renvoie vrai si la valeur de la propriété est plus grande que la valeur spécifiée
- **$gte** : renvoie vrai si la valeur de la propriété est plus grande ou égale que/à la valeur spécifiée

Par exemple, voici une recherche des produits ayant un prix supérieur ou égal à 300 euros et inférieur ou égal à 400 euros :

```
db.Products.find({"price" : {$gte:300, $lte:400}})
```

1.3.6.3 L'opérateur $exists

L'opérateur *$exists* permet de filtrer les documents qui contiennent ou pas la propriété spécifiée (quelque soit la valeur de celle-ci).

Par exemple, voici une recherche des produits de la collection *Products* qui sont associés à la propriété *popularity* :

```
db.Products.find({"popularity" : {$exists: true}})
```

1.3.6.4 L'opérateur *$in*

L'opérateur *$in* permet d'énumérer les valeurs possibles pour une propriété.

Par exemple, voici une recherche des produits de la collection *Products* dont le *type* est soit *computer*, soit *tablet* :

```
db.Products.find({"type":{$in: ["computer", "tablet"]}})})
```

1.3.6.5 L'opérateur *$nin*

L'opérateur *$nin* permet d'exclure une série de valeurs pour une propriété donnée.

Par exemple, voici une recherche des téléphones qui ne sont ni de marque *Konia*, ni de marque *Wiko* :

```
db.Products.find({"type":"phone", "brand":{$nin: ["Konia", "Wiko"]})
```

1.3.6.6 L'opérateur *$or*

L'opérateur *$or* permet de spécifier plusieurs conditions dont l'une au moins doit être vraie sur la même propriété.

Par exemple, voici une recherche des téléphones dont la popularité est soit maximale, soit non renseignée :

```
db.Products.find({"type":"phone", $or: [{"popularity":5},
                                        {"popularity": {$exists: false}}]
                 })
```

1.3.6.7 L'opérateur *$not*

L'opérateur *$not* permet de filtrer les documents qui ne satisfont pas la conditions spécifiée .

Par exemple, voici une recherche des *produits* qui n'aient pas un indice de popularité inférieur à 4 (cela inclut aussi les produits qui n'ont pas cet indice).

```
db.Products.find({"popularity": {$not: {$lt: 4}}})
```

1.3.6.8 L'opérateur $nor

L'opérateur *$nor* permet de filtrer les documents qui ne satisfont aucune des conditions spécifiées.

Par exemple, voici une recherche des *produits* qui ne sont ni des *téléphones* et ni ne coûtent plus de 300 euros.

```
db.Products.find({"$nor": [{"type":"phone"}, {"price":{$gte:500}}]})
```

1.3.7 L'application d'expressions régulières

Il est possible d'appliquer une expression régulière sur la valeur d'une propriété (par exemple pour y rechercher une sous-chaîne),

- soit directement en exprimant l'expression régulière entre deux barres obliques (suivant la légendaire syntaxe du langage Perl) ;
- soit en créant un objet avec la fonction constructrice **RegExp()**, notamment si l'expression régulière doit être construite via une variable.

Voici la recherche de la sous-chaîne "phone" comme valeur de la propriété *name* des documents de la collection *Products*.

```
db.Products.find ({"name": /phone/i})
```

```
var keyword = "phone"
db.Products.find({"name": new RegExp(keyword, "i")})
```

1.3.8 Les projections et la méthode *distinct()*

1.3.8.1 Les projections

Lors de l'utilisation de la méthode *find()*, un objet en second paramètre peut être donné pour circonscrire les propriétés des documents qui seront renvoyés.
Les propriétés de cet objet sont les noms des propriétés :

- soit à insérer dans les documents renvoyés si elles sont accompagnées de la valeur 1 ;

— soit à exclure des documents renvoyés si elles sont accompagnées de la valeur 0.

attention : il est impossible de simultanément conserver et exclure des propriétés (hormis pour exclure la propriété *_id* qui est renvoyée par défaut).

Par exemple, voici la recherche des noms des téléphones :

```
db.Products.find({"type": "phone"}, {"name": 1})
```

ou la recherche de toutes les informations sur les produits sauf leurs popularités :

```
db.Products.find({}, {"popularity": 0})
```

1.3.8.2 La méthode *distinct()*

La méthode *distinct()* renvoie les valeurs différentes d'une propriété des documents filtrés.

Les deux premiers paramètres de la méthode *distinct()* permettent de spécifier :

— la propriété dont seules les valeurs distinctes seront renvoyées ;
— l'objet filtre.

Par exemple, voici la recherche de toutes les *marques* différentes des produits vendus :

```
db.Products.distinct("brand")
```

ou encore, la recherche de toutes les marques différentes de téléphones :

```
db.Products.distinct("brand", {"type": "phone"})
```

1.3.9 Référencement des documents et jointures

Dans ce paragraphe, nous vous proposons de découvrir par des exemples le référencement de documents via des identifiants et la mise en œuvre de jointures.

Imaginons que dans le cadre de notre application de commerce en ligne, nous voulions gérer deux collections :

— la collection *Products* listant les produits mis en vente ;
— la collection *Carts* mémorisant les paniers (c'est à dire les commandes en constitution et non encore validées) de nos différents clients.
(des homonymes étant attendus chez nos clients, leurs adresses emails

nous permettront de les distinguer une fois leur identification sur notre site effectuée.)

Prenons en exemple les deux documents suivants de la collection *Products* :

```
{ "_id": ObjectId("58d3cd3cda8751c171a6606f"), "type":"phone",
  "brand":"Threestars", "name":"bigPhone", "price":200, "stock":25 }
{ "_id": ObjectId("58d3cd3cda8751c171a66071"), "type":"headset",
  "brand":"Earlid", "name":"Earlid Pro", "price":300, "stock":10 }
```

Ces deux produits doivent être ajoutés au panier de *Pierre Pompidor* dans un document de la collection *Carts*, et dans ce document en valeurs de la propriété *order*. Voici le document concernant le panier de *Pierre Pompidor* avant l'ajout des deux produits :

```
{ "_id": ObjectId("58d517d42576fcbc8bb9aa81"),
  "lastname": "Pompidor", "firstname": "Pierre", "email":"pompidor@lirmm.fr",
  "order" : [] }
```

Quelles sont les possibilités qui s'offrent à nous ? En fait, il y en a deux :

— imbriquer les objets correspondants aux produits dans l'objet panier ;
— référencer par leurs identifiants les objets produits dans l'objet panier.

1.3.9.1 Les objets imbriqués (*nested objects*)

La première solution est de recopier (totalement ou en partie) les propriétés des produits concernés dans des objets insérés dans la liste *order*.
Voici comment cela pourrait être incarné dans notre exemple :

```
{ "_id": ObjectId("58d517d42576fcbc8bb9aa81"),
  "lastname":"Pompidor", "firstname":"Pierre",
  "email":"pompidor@lirmm.fr",
  "order" : [
        {"type":"phone", "brand":"Threestars", "name":"bigPhone", "price":200},
        {"type":"headset", "brand":"Earlid", "name":"Earlid Pro", "price":200}
  ]
}
```

Voici la requête qui nous permet alors de connaître les noms des produits du panier de *Pierre Pompidor* :

```
db.Carts.find({"email":"pompidor@lirmm.fr"})[0]
        .order
        .forEach(function(doc){ print(doc.name); })
```

En conclusion, la duplication des données est critiquable, mais l'accès est relativement simple.

1.3.9.2 Les objets référencés

La seconde solution est de ne recopier que les identifiants des produits dans la liste des produits en commande :

```
{ "_id": ObjectId("58d517d42576fcbc8bb9aa81"),
  "lastname":"Pompidor", "firstname":"Pierre",
  "email":"pompidor@lirmm.fr",
  "order" : [
         ObjectId("58d3cd3cda8751c171a6606f"),
         ObjectId("58d3cd3cda8751c171a66071")
   ]
}
```

Faire afficher les noms de produits du panier va alors s'avérer un peu plus compliqué :

— soit nous allons devoir utiliser une variable intermédiaire qui contiendra la liste des identifiants des produits ;
— soit nous allons mettre en œuvre une jointure.

Voici la première solution, la seconde sera illustrée dans le paragraphe suivant.

```
var ids = db.Carts.find({"email":"pompidor@lirmm.fr"})[0].order
db.Products.find({_id: {$in: ids}}).forEach(function(doc) { print(doc.name); } )
```

En conclusion, le référencement des produits par leurs ids est plus élégant mais l'accès aux informations est plus laborieux.

1.3.9.3 Les jointures

A partir de la version 3.2 de MongoDB, il est possible d'effectuer une **jointure**.

La jointure est une action définie dans une suite d'actions (un *pipeline*) gérée par la méthode **aggregate()** appliquée sur une collection.
Voici le schéma de programmation d'un pipeline :

```
db.<nomDeLaCollection>.aggregate([
  {$<action> : },
  {$<action> : },
  ...
])
```

Les actions que nous allons mettre en œuvre dans notre exemple sont :

— **$match** : sélectionne les documents d'une collection ;
— **$unwind** : ditribue les éléments d'une liste sur l'objet qui la contient ;
— **$lookup** : applique une jointure.

Etape 1 :
La première action consiste à ne sélectionner dans la collection que le panier de

Pierre Pompidor.
Cette sélection est faite par l'instruction **$match** :

```
db.Carts.aggregate([
    {$match: {"email":"pompidor@lirmm.fr"}}
])
```

Voici le résultat des actions du pipeline :

```
{ "_id": ObjectId("58d533262576fcbc8bb9aa82"),
  "lastname":"Pompidor", "firstname":"Pierre",
  "email" : "pompidor@lirmm.fr",
  "order" : [ ObjectId("58d3cd3cda8751c171a6606f"),
              ObjectId("58d3cd3cda8751c171a66071")
            ]
}
```

Etape 2 :
Les identifiants des produits étant regroupés dans une liste, nous devons les distribuer sur l'objet contenant cette liste pour produire autant d'objets que de produits :

```
db.Carts.aggregate([
    { $match: {"email":"pompidor@lirmm.fr"}},
    { $unwind: "$order" }
])
```

Voici le résultat des actions du pipeline :

```
{ "_id" : ObjectId("58d55d616d577b19d5946c1d"),
  "lastname": "Pompidor", "firstname": "Pierre",
  "email": "pompidor@lirmm.fr", "order" : ObjectId("58d3cd3cda8751c171a6606f") }

{ "_id" : ObjectId("58d55d616d577b19d5946c1d"),
  "lastname": "Pompidor", "firstname": "Pierre",
  "email": "pompidor@lirmm.fr", "order" : ObjectId("") }
```

Etape 3 :
Nous allons maintenant appliquer la jointure (une *left join*) avec l'instruction **$lookup**.

Les propriétés de l'objet paramétrant la jointure vont indiquer :
- **from** : le nom de la collection sur laquelle est opérée la jointure ;
- **localField** : le nom de la propriété locale qui crée le lien de jointure ;
- **foreignField** : le nom de la propriété de la seconde collection qui crée le lien de jointure ;
- **as** : le nom de la nouvelle propriété qui va accueillir les documents.

```
db.Carts.aggregate([
  { $match:  {"email":"pompidor@lirmm.fr"}},
  { $unwind: "$order" },
  { $lookup: { from: "Products",
               localField: "order",
               foreignField: "_id",
               as: "product" }}
])
```

Voici le résultat des actions du pipeline :

Le premier objet produit :

```
{ "_id" : ObjectId("58d55d616d577b19d5946c1d"),
  "lastname": "Pompidor", "firstname": "Pierre", "email" : "pompidor@lirmm.fr",
  "order" : ObjectId("58d3cd3cda8751c171a6606f"),
  "product" : [ { "_id" : ObjectId("58d3cd3cda8751c171a6606f"),
                  "type": "phone", "brand": "Threestars", "name": "bigPhone",
                   "popularity": 4, "price": 200, "picture": "bigphone.jpeg",
                  "stock": 25 }
              ]
}
```

Et le second objet produit :

```
{ "_id" : ObjectId("58d55d616d577b19d5946c1d"),
  "lastname": "Pompidor", "firstname": "Pierre", "email" : "pompidor@lirmm.fr",
  "order" : ObjectId("58d3cd3cda8751c171a66071"),
  "product" : [ { "_id" : ObjectId("58d3cd3cda8751c171a66071"),
                  "type": "headset", "brand": "Earlid", "name": "Earlid Pro",
                  "popularity": 5, "price": 300, "picture": "beatspro.jpeg",
                  "stock": 21 }
              ]
}
```

Etape 4 :

Si nous voulons revenir à une structure assez semblable de notre panier qui comprenait une propriété *order* contenant la liste des identifiants des produits sélectionnés à laquelle nous rajoutons une propriété *products* qui contiendra les objets produits, voici le pipeline que nous exécutons :

```
db.Carts.aggregate([
     { $match: {"email":"pompidor@lirmm.fr"}},
     { $unwind: "$order" },
     { $lookup: { from: "Products",
                  localField: "order",
                  foreignField: "_id",
                  as: "product" }},
     { "$unwind": "$product" },
     { "$group": {"_id": "$_id",
                  "order": { "$push": "$order" },
                  "products": { "$push": "$product" }
                 }
     }
]
)
```

Voici le résulats des actions du pipeline :

```
{ "_id" : ObjectId("58d55d616d577b19d5946c1d"),
  "order" : [ ObjectId("58d55cd4dca60e1fe1f94d61"), ObjectId("58d55cd4dca60e1fe1f94d63") ],
  "products" : [
    { "_id": ObjectId("58d55cd4dca60e1fe1f94d61"),
      "type": "phone", "brand": "Threestars", "name": "bigPhone",
      "popularity" : 4, "price": 200, "picture": "bigphone.jpeg", "stock": 25 },
    { "_id": ObjectId("58d55cd4dca60e1fe1f94d63"),
      "type": "headset", "brand" : "Earlid", "name" : "Earlid Pro",
      "popularity": 5, "price": 300, "picture": "beatspro.jpeg", "stock": 21 }
  ]
}
```

1.3.10 Mise à jour et suppression d'un document

1.3.10.1 Mise à jour d'un document

La mise à jour d'un document est mise en œuvre grâce à la méthode *update()* qui prend deux paramètres :

- l'objet filtre
- un objet regroupant les modifications à effectuer :
  - la modification de la valeur d'une propriété existante, ou l'ajout d'une nouvelle propriété, sont introduites par l'opérateur $set
  - la suppression d'une propriété est introduite par l'opérateur $unset

```
db.<NomDeLaCollection>.update(<ObjetFiltre>, <ObjetSpécifiantLesMisesAJour>)
```

Par exemple, nous voulons modifier le prix de *l'topPhone 7 64G* pour le baisser vertigineusement de 1000 euros à 999 euros, nous écrirons :

```
db.Products.update({"name": "topPhone 7 64G"}, {$set: {"price": 999}})
```

Il est à noter que la requête suivante :

```
db.Products.update({"name": "topPhone 7 64G"}, {"price": 999})
```

n'aurait pas du tout effectué ce que nous voulions faire :
toutes les propriétés du document sélectionné auraient été supprimées et seule la nouvelle (*"price" : 999*) inscrite dans l'objet.

Par ailleurs, attention, notre modification ne concernera que le premier document satisfaisant les critères. Si vous voulez que tous les documents satisfaisant les critères soient modifiés, vous devez :

soit rajouter l'opérateur *$multi* avec la valeur *true* :

```
db.Products.update({"type": "tablet"}, {$set: {"type": "touch tablet"}, "multi": true})
```

soit utiiliser la méthode *updateMany()*.

1.3.10.2 Suppression d'un document

La suppression d'un ou plusieurs documents est mise en œuvre grâce à la méthode *remove()* :

```
db.<NomDeLaCollection>.remove(<ObjetFiltre>)
```

1.3.11 Suppression d'une collection

La suppression d'une collection est mise en œuvre grâce à la méthode *drop()* :

```
db.<NomDeLaCollection>.drop()
```

1.4 Utilisation de MongoDB via Node.js

Pour illustrer l'association entre *Node.js* et *MongoDB* nous allons créer différentes versions d'un serveur Node.js nommé *onlinesalesserver*.
Ce serveur sera configuré par le fichier *package.json*.

1.4.1 Installation du module MongoDB pour Node.js

La première étape est de créer le squelette du fichier *package.json* avec la commande `npm init`.
Par rapport aux questions demandées par cette commande, nous ne renseignons que le nom du serveur, sa description, le fichier qui le contient et notre nom.
Voici le fichier *package.json* ainsi produit :

```
{
{
  "name": "onlinesalesserver",
  "version": "1.0.0",
  "description": "serveur de vente de produits hitech",
  "main": "onlinesalesserver.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Pierre Pompidor",
  "license": "ISC"
}
```

Comme nous avons installé Node.js globalement, nous interrogeons sa version avec `node -v`, et nous complétons ce fichier avec la propriété *engines*.
Nous complétons aussi l'objet *scripts* avec la propriété *start* ce qui permet d'utiliser *npm* pour lancer le serveur :

```
{
  ...
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node onlinesalesserver.js"
  },
  ...
  "engines": { "node": ">= 7.6.0" }
}
```

La seconde étape est d'installer les modules **express**, **cors** et bien sûr *mongodb* :

```
npm install express cors mongodb --save
```

L'option *–save* permet de compléter les dépendances listées dans le fichier *package.json*. voici donc la version (pour l'instant) finale de ce fichier.

```
{
  "name": "onlinesalesserver",
  "version": "1.0.0",
  "description": "serveur de produits hitech",
  "main": "onlinesalesserver.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node onlinesalesserver.js"
  },
  "author": "Pierre Pompidor",
  "license": "ISC",
  "engines": { "node": ">= 7.6.0" },
  "dependencies": {
    "cors": "^2.8.3",
    "express": "^4.15.3",
    "mongodb": "^2.2.28"
  }
}
```

1.4.2 Connexion au serveur MongoDB

Pour ce connecter au serveur *MongoDB* à partir de *Node.js*, nous allons :
— créer une application cliente MongoDB grâce au module *mongodb* de *Node.js* ;
— nous connecter au serveur MongoDB grâce à la méthode *connect()* qui va prendre comme paramètres :

- l'url de connexion : le protocole réseau, l'adresse IP du serveur MongoDB et la base de données (dans notre cas *OnlineSales*) :
- une fonction de callback qui porte deux paramètres :
  - une variable qui aura la valeur *null* si la connexion a réussi ou sinon l'erreur de connexion ;
  - un objet qui permettra la connexion à la base de données sélectionnée (si bien sûr la connexion a réussi).

Soit le programme *onlinesalesserver.js* suivant :

```
"use strict";

var MongoClient = require("mongodb").MongoClient;
var assert = require("assert");
var url = "mongodb://localhost:27017/Onlinesales";

MongoClient.connect(url, {useNewUrlParser: true}, function(err, db) {
   assert.equal(null, err);
   console.log("Connexion au serveur MongoDB réussi");
   db.close();
});
```

Si nous exécutons ce serveur via l'instruction `npm start` (nous pourrions bien sûr l'exécuter directement avec la commande `node onlinesalesserver.js`), le message de connexion apparaîtra dans votre terminal ou votre invite de commandes.

(Le serveur *onlinesalesserver.js* évoluant suivant les exemples présentés, cette version est sauvegardée sous le nom de *testConnexionAuServeurMongoDB.js*).

1.4.3 Insertion de données à partir d'un serveur Node.js

Deux méthodes *insertOne()* et *insertMany()* permettent d'insérer des documents dans une collection MongoDB. Si la collection n'existe pas, elle est créée.

Voici le schéma de programmation permettant d'insérer un seul document avec la méthode *insertOne()* :

```
db.collection("<nom de la collection>").insertOne( { ... } );
```

Voici le schéma de programmation permettant d'insérer plusieurs documents avec la méthode *insertMany()* :

```
db.collection("nom de la collection").insertMany( [ ... ] );
```

Voici un exemple d'insertion d'un document dans la collection *Products* :

```
MongoClient.connect(url, {useNewUrlParser: true}, function(err, db) {
   assert.equal(null, err);
   db.collection("Products").insertOne( {
      "type":"phone", "brand":"Peach", "name":"topPhone 7 32G",
      "popularity":4, "price":900, "picture":"topphone.jpeg", "stock":5 } );
   db.close();
})
```

1.4.4 Interrogation de données à partir d'un serveur Node.js

Après la connexion au serveur MongoDB, nous allons effectuer différentes sélections sur les documents contenus dans la collection *Products* de notre base de données *Onlinesales*.
Pour que nos codes soient plus lisibles, ces différentes sélections seront isolées dans une fonction *productResearch()*.

1.4.4.1 Exploitation du résultat de la méthode *find()*

La méthode *find()* renvoie un objet *Promise* sur lequel est appliqué la méthode *each()* qui nous permet d'accéder à chaque document.
Soit le code du serveur qui affiche tous les documents de la collection *Products* :

```
"use strict";
var MongoClient = require("mongodb").MongoClient;
var assert = require("assert");
var url = "mongodb://localhost:27017/OnlineSales";

var productResearch = function(db) {
    var cursor = db.collection("Products").find();
    cursor.each(function(err, doc) {
        assert.equal(err, null);
        if (doc != null)
           for (let p in doc) console.log(p+" : "+doc[p]);
        console.log("\n");
   });
};

MongoClient.connect(url, {useNewUrlParser: true}, function(err, db) {
   assert.equal(null, err);
   productResearch(db);
   db.close();
});
```

Si ce code est exécuté via l'instruction `npm start`, les propriétés/valeurs de chaque document de la collection *Products* sont affichées dans votre terminal ou votre invite de commandes.

```
_id : 58b3e2d440f60da0834601da
type : phone
brand : Peach
name : topPhone 7 32G
popularity : 4
price : 900
photo : topphone.jpeg
...
```

(Cette version du serveur est sauvegardée sous le nom de *interrogationMongoDB_ affichageConsole.js*).

Essayons maintenant d'afficher un message une fois terminé l'affichage dans la console des documents sélectionnés. Mettons également en œuvre un objet faisant un filtre sur les produits de type *phone*.

```
"use strict";
var MongoClient = require("mongodb").MongoClient;
var assert = require("assert");
var url = 'mongodb://localhost:27017/OnlineSales';

var productResearch = function(db, ObjetFiltre, callback) {
   var cursor = db.collection("Products").find(ObjetFiltre);
   cursor.each(function(err, doc) {
        assert.equal(err, null);
        if (doc != null)
            for (let p in doc) console.log(p+' : '+doc[p]);
        else console.log("Fin du traitement");
        console.log("\n");
   });
};

MongoClient.connect(url, {useNewUrlParser: true}, function(err, db) {
   assert.equal(null, err);
   productResearch(db, {"type": "phone"});
   db.close();
});
```

(Cette version du serveur est sauvegardée sous le nom de *interrogationMongoDB_ affichageConsole.js*).

1.4.4.2 Utilisation de la méthode toArray()

Il est vraiment pratique d'utiliser la méthode *toArray()* qui stocke les documents renvoyés par la requête dans une liste sur le résultat de *find()*.
En voici un exemple où le nom des produits stockés dans la collection *Products* sont affichés dans la console :

```
'use strict';
var MongoClient = require('mongodb').MongoClient;
var assert = require('assert');
var url = 'mongodb://localhost:27017/OnlineSales';

var findProduits = function(db, search) {
    db.collection('Products').find().toArray(function(err, documents) {
       for (let doc of documents) console.log(doc.name);
    });
};

MongoClient.connect(url, {useNewUrlParser: true}, function(err, db) {
   assert.equal(null, err);
   findProduits(db);
   db.close();
});
```

(Cette version du serveur est sauvegardée sous le nom de *interrogationMongoDB_ toArray.js*).

Voici un exemple où nous mettons en œvre un objet faisant un filtre sur les produits de type *phone* et de marque *Peach* (un message est par ailleurs affiché une fois terminé l'affichage des documents sélectionnés).

```
var MongoClient = require('mongodb').MongoClient;
var assert = require('assert');
var url = 'mongodb://localhost:27017/OnlineSales';

var findProduits = function(db, objetFiltre, callback) {
   var cursor = db.collection('Products').find(objetFiltre);
    cursor.each(function(err, document) {
        assert.equal(err, null);
        if (document != null)
           for (let prop in document) console.log(prop+" : "+document[prop]);
        else { callback(); }
        console.log("\n");
   });
};

MongoClient.connect(url, {useNewUrlParser: true}, function(err, db) {
   assert.equal(null, err);
    findProduits(db, {"type": "phone", "brand": "Peach"}, function() {
       console.log("Fin de l'interrogation");
   });
   db.close();
});
```

(Cette version du serveur est sauvegardée sous le nom de *interrogationMongoDB_filtre.js*).

1.4.5 La synchronisation des requêtes

Dans l'univers de JavaScript, les traitements d'importations et d'interrogations de données (que ce soit via *Ajax* ou via des requêtes sollicitant une base de données NoSQL comme *MongoDB*) sont par défaut **asynchrones**. Ce modèle, naturel avec la gestion événementielle du langage, est très puissant car il évite de figer les pages web, et les actions de l'utilisateur, en attendant que les données soient importées. Mais, cela-dit, nous rencontrons des cas où notre souhait est à l'inverse de synchroniser des accès aux données, et la programmation de ces cas-là mérite toute notre attention.

Pour illustrer notre propos, imaginons que nous désirons sélectionner les téléphones de notre boutique en ligne qui sont plus chers que la moyenne de tous les téléphones que nous vendons.
Imaginons également que nous ne savons pas réaliser cette interrogation en une seule requête, mais que nous savons déjà comment seront paramétrées nos deux requêtes.

Nous allons dans un premier temps filtrer notre collection *Products* par l'objet filtre suivant :

```
{"type" : "phone"}
```

puis stocker dans une variable (nommée *average*) la moyenne des prix des téléphones que nous vendons, puis injecter ce résultat dans le second objet filtre paramétrant notre seconde requête qui interrogera de nouveau notre collection *Products* :

```
{"price" : {$gte: average}}
```

Pour unifier le résultat de nos requêtes, nous créons notre propre fonction *productResearch()* qui reçoit en paramètres :

- le lien sur la base de données
- un objet composé de :
  - du message à faire afficher une fois que la sélection des documents aura été faite ;
  - l'objet filtre.

puis exécute la requête grâce à la méthode *find()* du module *mongodb*, et renvoie dans la fonction de callback de la méthode *toArray()* :

- le message à afficher de fin de traitement ;
- et la collection des documents sélectionnés (ou une collection vide si aucun document n'a été sélectionné).

```
function productResearch(db, param, callback) {
   db.collection("Products").find(param["filterObject"])
     .toArray(function(err, documents) {
        if (err) callback(err, []);
        if (documents !== undefined) callback(param["message"], documents);
        else callback(param["message"], []);
    });
};
```

1.4.5.1 L'utilisation des fonctions de callback

La solution naturelle pour synchroniser des requêtes à la base de données NoSQL est d'encapsuler les requêtes dans les fonctions de callback.
Voici une implémentation possible de notre exemple :

```
"use strict";
var MongoClient = require("mongodb").MongoClient;
var assert = require("assert");
var url = "mongodb://localhost:27017/OnlineSales";

... // function productResearch

MongoClient.connect(url, {useNewUrlParser: true}, function(err, db) {
  assert.equal(null, err);

  find(db, {"message":"Etape 1", "ObjetFiltre": {"type" : "phone"}},
       function(etape, resultats) {
     console.log(etape+" avec "+resultats.length+" produits sélectionnés");
     if (resultats.length > 0) {
        let average = 0;
        for (let document of resultats)
           if (document.price !== undefined)  average += document.price; }
           average /= resultats.length;
           console.log("average = "+average);
           find(db, {"message":"Etape 2", "ObjetFiltre": {"price" : {$gte: average}}},
                function(etape, resultats) {
           console.log(etape+" avec "+resultats.length+" produits sélectionnés");
           for (let document of resultats)
               console.log(document.name+" : "+document.price);
           db.close();
        });
     }
     else db.close();
  });
});
```

Voici le résulat par rapport à notre collection *Products* :

```
Etape 1 avec 5 produits sélectionnés
moyenne = 780
Etape 2 avec 4 produits sélectionnés
topPhone 8 32G : 900
topPhone 8 64G : 1000
topPhone 8 256G : 1300
Peach pro : 1300
```

Cette première solution d'implémentation est satisfaisante mais si nous devions enchaîner beaucoup de requêtes à synchroniser, elle deviendrait problématique au niveau de la lisibilité du programme.
Nous pouvons alors être précipîtés dans l'enfer des callbacks (The Callback hell) dont voici un exemple vertigineux :

```
db.<NomDeLaCollection>.find(db, <ObjetFiltre>, function(message, results) {
  ...
  db.<NomDeLaCollection>.find(db, <ObjetFiltre>, function(message, results) {
    ...
    db.<NomDeLaCollection>.find(db, <ObjetFiltre>, function(message, results) {
      ...
      db.<NomDeLaCollection>.find(db, <ObjetFiltre>, function(message, results) {
        ...
      });
    });
  });
});
```

1.4.5.2 L'utilisation du module *async*

Pour éviter "l'enfer des callbacks", nous pouvons utiliser un module très puissant de gestion de la (a)synchronisme des traitements exécutés sous *Node.js* : le module **async**.

Pour installer ce module et le référencer dans le fichier *package.json*, exécutez la commande suivante : `npm install --save async`.

Parmi les nombreuses méthodes que propose ce module, deux permettent de synchroniser des requêtes :
— la méthode *series()*
— la méthode *waterfall()*

Les deux méthodes permettent de lister les fonctions à enchaîner, mais les données qui peuvent être passées de d'une à une autre sont gérées différemment.
Dans les exemples suivants, la connexion à la base de données MongoDB et la fonction *productResearch()* ne seront pas reproduites.

La méthode *async.series()* suit la structure suivante :

```
// déclarations des données à partager entre les fonctions
...
async.series([
  function (callback) {  // première fonction
    callback();
  },
  function (callback) {  // seconde fonction
    callback();
  },
   ...
  ],
  function() { ... }  // fonction finale
);
```

Les données à partager entre les fonctions doivent être déclarées au préalable. Un appel `callback(true)` arrête l'enchaînement des fonctions et appelle la fonction finale si elle existe.

La méthode *async.waterfall()* suit la structure suivante :

```
async.waterfall([
  function (callback) {  // première fonction
    callback(null, parametre1, ...);
  },
  function (parametre1, ..., callback) {  // seconde fonction
    callback(null);
  },
   ...
  ],
  function() { ... }  // fonction finale
);
```

Les données à partager entre les fonctions sont passées en paramétres, c'est à dire que chaque fonction les transmet à la fonction suivante via l'appel de la fonction de callback. Un appel `callback(true)` arrête l'enchaînement des fonctions et appelle la fonction finale si elle existe.

1.4.5.3 La méthode *async.series()*

Voici le fragment de notre exemple de code utilisant la méthode *async.series()* :

```
var async = require("async");
MongoClient.connect(url, function(err, db) {
  assert.equal(null, err);
  var average = 0;

  async.series([
    function (callback) {
      productResearch(db, {"message":"Etape 1",
                      filterObject": {"type":"phone"}},
                      function(message, results) {
          console.log(message+" : "+results.length+" produits sélectionnés");
          if (results.length > 0) {
             for (let document of results) average += document.price;
                average /= results.length;
                console.log("moyenne = "+average);
                callback();
             }
           else callback(true);
       });
    },
    function (callback) {
       productResearch(db, {"message":"Etape 2",
                           "filterObject": {"price":{$gte: average}}},
                        function(message, results) {
                console.log(message+" : "+results.length+" produits sélectionnés");
                for (let doc of results) console.log(doc.name+" : "+doc.price);
                callback();
       });
    }],
    function () {  db.close(); }
  );
});
```

Voici le résulat par rapport à notre collection *Products* :

```
Etape 1 : 5 produits sélectionnés
moyenne = 780
Etape 2 : 4 produits sélectionnés
topPhone 8 32G : 900
topPhone 8 64G : 1000
topPhone 8 256G : 1300
Peach pro : 1300
```

1.4.5.4 La méthode *async.waterfall()*

Voici le fragment de notre exemple de code utilisant la méthode *async.waterfall()* :

```
...
var async = require("async");

MongoClient.connect(url, function(err, db) {
    assert.equal(null, err);
    async.waterfall([
      function(callback) {
         productResearch(db, {"message":"Etape 1",
                         "filterObject": {"type" : "phone"}},
                         function(message, results) {
            console.log(message+" : "+results.length+" produits sélectionnés");
            if (results.length > 0) {
                var average = 0;
                for (let doc of results) if (doc !== undefined) average += doc.price;
                average /= results.length;
                console.log("moyenne = "+average);
                callback(null, average);
             }
             else callback(true);
         });
      },
      function(average, callback) {
         productResearch(db, {"message":"Etape 2",
                              "filterObject": {"price" : {$gte: average}}},
                              function(message, results) {
             console.log(message+" : "+results.length+" produits sélectionnés");
             for (let doc of results) console.log(doc.name+" : "+doc.price);
             callback(null);
         });
      }],
    function() { db.close(); }
  );
});
```

Voici le résulat par rapport à notre collection *Products* :

```
Etape 1 : 5 produits sélectionnés
moyenne = 780
Etape 2 : 4 produits sélectionnés
topPhone 8 32G : 900
topPhone 8 64G : 1000
topPhone 8 256G : 1300
Peach pro : 1300
```

1.5 Interrogation de MongoDB via les routes gérées par *express*

Nous allons maintenant interroger notre base de données MongoDB en invoquant le serveur Node.js par des requêtes HTTP qui expriment des routes. Ces routes sont gérées par le module *express* de *Node.js*.

1.5.1 Structure d'un serveur Node.js interrogeant MongoDB

Un serveur *Node.js* interrogeant MongoDB aura la structure suivante :
- Utilisation du module *express* :
  - pour créer une application gérant les routes
  - et qui écoute sur un port donné
- Utilisation du module *cors* pour permettre le Cross-Origin Resource Sharing
- Création du client MongoDB qui
  - se connecte à une base de données gérée par le serveur MongoDB
  - mise en place des "écouteurs" sur les différentes routes

Dans l'exemple structurel suivant, `<route>` désigne une route quelconque.

```
"use strict";
var express=require("express");  // Utilisation du module Express
var app=express();               // pour créer une application Express
app.listen(8888);                // qui écoute sur un port donné
var cors=require("cors");        // Utilisation du module CORS
app.use(cors());                 // pour permettre le Cross-Origin Resource Sharing

var MongoClient = require("mongodb").MongoClient;
var url = "mongodb://localhost:27017/Onlinesales";
var assert = require("assert");

MongoClient.connect(url, function(err, db) {
    assert.equal(null, err);
    app.get(<route>, function(req, res){  // gestion d'une route (méthode GET)
      ...
    });
    ... // autres gestions de routes via la méthode GET
    app.post(<route>, function(req, res){  // gestion d'une route (méthode POST)
      ...
    });
    ... // autres gestions de routes via la méthode POST
    ... // gestions des routes pour les méthodes PUT et DELETE

});
```

1.5.2 La problématique du Cross-Origin Resource Sharing

Le **Cross-Origin Resource Sharing** implique qu'une ressource demandée par le client doit être donnée par un serveur qui n'est pas celui qui a servi la page web qui demande cette ressource. Par défaut un mécanisme de sécurité empêche le navigateur d'accéder à cette ressource.

Nous ne détaillerons pas cette problématique, mais pour lever cette sécurité nous vous proposons au niveau du serveur Node.js de mettre en œuvre le module *cors* via le code suivant :

```
var cors=require("cors");  // Utilisation du module CORS
app.use(cors());           // pour permettre le Cross-Origin Resource Sharing
```

et par ailleurs de modifier l'entête HTTP des réponses envoyées au client par l'instruction suivante (*res* étant l'objet "réponse") :

```
res.setHeader('Access-Control-Allow-Origin', '*');
```

1.5.3 Exemples de gestion de routes

Tous les exemples de gestions de routes que nous allons voir dans ce paragraphe portent sur la collection *Products* qui regroupent les produits mis en vente.
Ces codes vont constituer le socle du serveur de notre application de e-commerce.

Les exemples que nous présentons dans les paragraphes suivants vont concerner des recherches appliquées sur notre collection de produits via des critères de recherche (hormis l'identifiant interne attribué par mongoDB à chaque document de la collection).
Pour structurer notre code, nous utilisons de nouveau la fonction *productResearch()* qui :

- reçoit en paramètre un objet contenant :
  - le message à faire afficher en fin de sélection ;
  - l'objet spécifiant les critères de recherche sur la collection *Products*.
- et appelle la fonction de callback avec :
  - le message à afficher (qui peut être remplacé par un message d'erreur) ;
  - la liste des documents filtrés.

```
function productResearch(db, param, callback) {
    db.collection("Products").find(param["filterObject"])
      .toArray(function(err, documents) {
        if (err) callback(err, []);
        else if (documents !== undefined) callback(param["message"], documents);
     else callback(param["message"], []);
    });
});
```

1.5.3.1 Gestion d'une route pour lister les marques

Imaginons que dans le cadre de notre fil rouge (notre application d'e-commerce), nous voulions gérer une route pour lister toutes les marques des produits que nous vendons. Ce service sera important pour que nos futurs clients puissent définir leurs critères de recherches.

Voici donc le code gérant la route `/Products/brands` :

```
MongoClient.connect(url, function(err, db) {
    assert.equal(null, err);

    app.get("/Products/brands",function(req,res){
        productResearch(db, {"message":"/Products/brands", "filterObject":{}},
                        function(etape, results) {
            console.log(etape+" : "+results.length+" products sélectionnés :");
            var brands = [];
            for (let doc of results)
                if (!brands.includes(doc.brand)) brands.push(doc.brand);
            marques.sort();
            var json=JSON.stringify(brands);
            console.log(json);
            res.setHeader("Content-type","application/json; charset=UTF-8");
            res.end(json);
        });
    });
});
```

1.5.3.2 Gestion d'une route pour filtrer les produits

Nous voulons maintenant gérer une route beaucoup plus ambitieuse :
- toujours sur la collection *Products*
- mais qui présentera, dans cet ordre, 5 paramètres définissant :
  - le type des produits recherchés : `:type`
  - la marque des produits recherchés : `:brand`
  - l'intervalle de prix des produits recherchés : `:minprice` et `:maxprice`
  - et la popularité minimale des produits recherchés : `:minpopularity`

Par exemple, voici donc, une fois fois la connexion à la base de données réussie, la gestion de la route filtrant les documents de la collection *Products*.
Le route est `/Products/:type/:brand/:minprice/:maxprice/:minpopularity`.

```
MongoClient.connect(url, function(err, db) {
    assert.equal(null, err);

    app.get("/Products/:type/:brand/:minprice/:maxprice/:minpopularity",
            function(req,res){
        var filterObject = {};
        if (req.params.type != "*") { filterObject.type = req.params.type; }
        if (req.params.brand != "*") { filterObject.brand = req.params.brand; }
        if (req.params.minprice != "*" || req.params.maxprice != "*") {
            filterObject.price = {};
            if (req.params.minprice != "*")
                filterObject.price.$gte = parseInt(req.params.minprice);
            if (req.params.maxprice != "*")
                filterObject.price.$lte = parseInt(req.params.maxprice);
        }
        if (req.params.minpopularity != "*") {
            filterObject.popularity = {$gte: parseInt(req.params.minpopularity)};
        }

        productResearch(db, {"message":"/Products",
                        "filterObject": filterObject}, function(etape, results) {
            console.log(etape+" avec "+results.length+" produits sélectionnés :");
            res.setHeader("Content-type","application/json; charset=UTF-8");
            var json=JSON.stringify(results);
            console.log(json);
            res.end(json);
        });
     });
});
```

1.5.3.3 Recherche d'un produit à partir de son identifiant interne

Dans le cadre de notre application, il nous sera aussi bientôt utile de pouvoir retrouver un produit à partir de son identifiant.
MongoDB associe chaque document d'une collection à un identifiant interne en valeur de la propriété `_id`.
Cet identifiant est codé sous la forme d'une chaîne de 24 caractères représentant chacun une valeur hexadécimale. La fonction *ObjectId()* fournie par le module *mongodb* crée les 12 octets correspondants. Pour l'utiliser dans notre exemple, nous créerons le raccourci suivant :

```
let ObjectId = require("mongodb").ObjectId;
```

Dans le code proposé ci-dessous, il est à remarquer que si l'identifiant est mal formé ou inconnu dans la collection, un objet vide est renvoyé.

```
app.get("/Product/id=:id", function(req,res) {
    let id = req.params.id;
    console.log("Dans /Product/id="+id);
    if (/[0-9a-f]{24}/.test(id))
        db.collection("Products").find({"_id": ObjectId(id)}).toArray(function(err, d
   let json = JSON.stringify({});
   if ( documents !== undefined && documents[0] !== undefined )
             json = JSON.stringify(documents[0]);
          console.log(json);
          res.end(json);
});
    else res.end(JSON.stringify({}));
});
```

Par exemple, la route :

```
localhost:8888/Product/id=58e643742ba15f90d13cb87d
```

renverra le document correspondant :

```
{"_id":"58e643742ba15f90d13cb87d","type":"phone",
 "brand":"Peach","name":"topPhone 8 32G","popularity":4,
 "price":900,"picture":"topphone.jpeg","stock":5}
```

Index

$eq (MongoDB), 12
$exists (MongoDB), 13
$gt (MongoDB), 12
$gte (MongoDB), 12
$in (MongoDB), 13
$lookup (MongoDB), 17, 18
$lt (MongoDB), 12
$lte (MongoDB), 12
$match (MongoDB), 17, 18
$ne (MongoDB), 12
$nin (MongoDB), 13
$nor (MongoDB), 14
$not (MongoDB), 13
$or (MongoDB), 13
$unwind (MongoDB), 17

aggregate() (MongoDB), 17
async (Node.js), 30
async.series() (Node.js), 32
async.waterfall() (Node.js), 33

BSON, 5

CORS, 35
count() (MongoDB), 11

Date() (MongoDB), 10
distinct() (MongoDB), 15
drop() (MongoDB), 22

each() (Node.js + MongoDB), 25
Expressions régulières (MongoDB), 14

find() (MongoDB), 11
forEach() (MongoDB), 11

getIndexes() (MongoDB), 7

Index (MongoDB), 7
insertMany() (MongoDB), 24
InsertMany() (Noed.js + MongoDB), 24
insertOne() (MongoDB), 24
InsertOne() (Noed.js + MongoDB), 24

Jointure (MongoDB), 17
JSON, 5

mongo shell (MongoDB), 8
mongoimport (MongoDB), 10

NoSQL, 5

ObjectId() (MongoDB), 37

print() (MongoDB), 8
Projection (MongoDB), 14

Redis, 5
RegExp (MongoDB), 14
remove() (MongoDB), 22

show collections (MongoDB), 9
show dbs (MongoDB), 9
show tables (MongoDB), 9

toArray() (Node.js + MongoDB), 12, 26