

## *Introduction à MongoDB*

Stéphane Malandain – Printemps 2024

---

### 1. Présentation

*MongoDB* est une base de données NoSQL distribué de type *Document Store* ([site web](#))

Objectifs :

- Gérer de gros volumes
- Facilité de déploiement et d'utilisation
- Possibilité de faire des choses complexes tout de même

### 2. Modèle des données

Principe de base : les données sont des documents

- stocké en *Binary JSON* (BSON)
- documents similaires rassemblés dans des collections
- pas de schéma des documents définis en amont
  - contrairement à un BD relationnel ou NoSQL de type *Column Store*
- les documents peuvent n'avoir aucun point commun entre eux
- un document contient (généralement) l'ensemble des informations
  - pas (ou très peu) de jointure à faire
- BD respectant **CP** (dans le théorème *CAP*)
  - propriétés ACID au niveau d'un document

### 3. JSON

- JavaScript Object Notation
- Créé en 2005
- On parle de **littéral**
- Format d'échange de données structurées léger
- Schéma des données non connu
  - contenu dans les données
- Basé sur deux notions :

- collection de couples clé/valeur
  - liste de valeurs ordonnées
- Structures possibles :
  - objet (couples clé/valeur) :
  - {}
  - {"nom": "jollois", "prenom": "fx" }
  - tableau (collection de valeurs) :
  - []
  - [ 1, 5, 10]
  - une valeur dans un objet ou dans un tableau peut être elle-même un littéral
- Deux types atomiques (string et number) et trois constantes (true, false, null)

Validation possible du JSON sur [jsonlint.com/](https://jsonlint.com/)

```

1  {
2      "isc125": {
3          "formation": "ISC SBD ",
4          "responsable": { "nom": "Martin", "prenom": "Marc" },
5          "etudiants" : [
6              { "id": 1, "nom": "Dupont", "prenom": "christophe" },
7              { "id": 2, "nom": "Durand", "details": "délégué" },
8              { "id": 5, "nom": "Walsh" }
9          ],
10         "ouverte": true
11     },
12     "isc652": {
13         "formation": "ISC P00n",
14         "ouverte": false,
15         "todo": [
16             "Creation de la maquette",
17             "Validation par le conseil"
18         ],
19         "responsable": { "nom": "" }
20     }
21 }
22

```

## 4. Compléments

BSON : extension de JSON

- Quelques types supplémentaires (identifiant spécifique, binaire, date, ...)
- Distinction entier et réel

### Schéma dynamique

- Documents variant très fortement entre eux, même dans une même collection
- On parle de **self-describing documents**
- Ajout très facile d'un nouvel élément pour un document, même si cet élément est inexistant pour les autres

- Pas de ALTER TABLE ou de redesign de la base

Pas de jointures entre les collections

## 5. Langage d'interrogation

- Pas de SQL (bien évidemment), ni de langage proche
- Définition d'un langage propre (basé sur JS)
- Langage permettant plus que les accès aux données
  - définition de variables
  - boucles
  - ...

## 6. Acces GUI studio3T (ex. Robo 3T)

<https://robomongo.org/>

studio3t.com/fr/download/

3T Produit Solutions Ressources Contactez-Nous Magasin FR Essai Gratuit S'adresser Au Service Des Ventes Mon 3T

# Télécharger Studio 3T

L'expérience ultime de l'interface graphique depuis 2014

Télécharger pour Apple Intel Télécharger pour Apple Silicon Autres plateformes

Studio 3T 2024.2.0 (13 mars 2024)

Notes de mise à jour | Sommes de contrôle SHA-256 | Politique de confidentialité | CLUF | Liste des modifications

“ En termes d'impact sur nos coûts d'exploitation, Studio 3T nous permet de gagner énormément de temps. Si nous n'avions pas Studio 3T, cela prendrait probablement dix fois plus de temps. ”

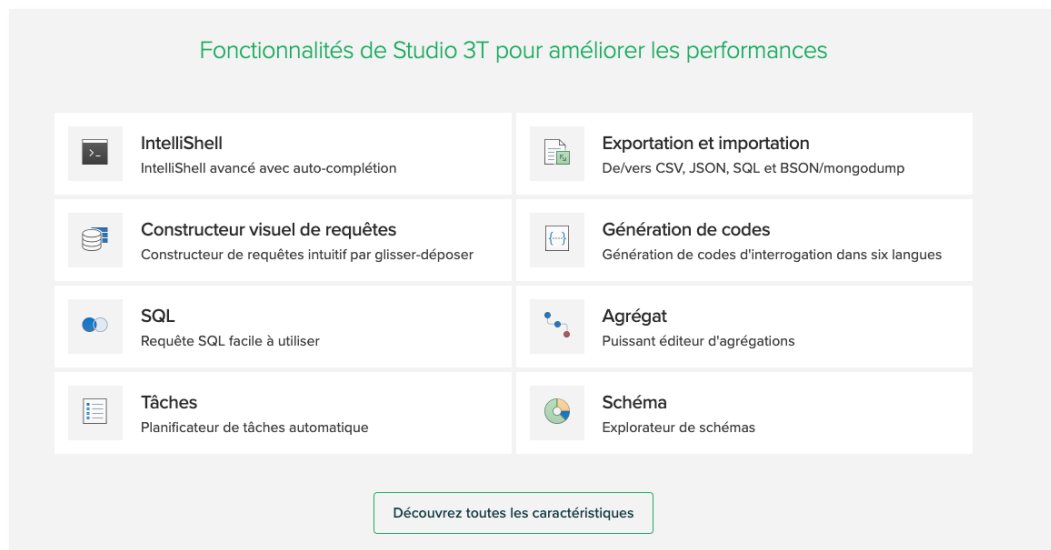
DATADOCK SOLUTIONS

Martin Adamec  
Directeur technique, Datadock Solutions

“ Le temps gagné par l'ensemble de l'organisation grâce à Constructeur visuel de requêtes vaut largement le prix des licences ! ”

#FloQast

Eric Hirshfield-Yamanishi  
SDET, FloQast



## 7. Installation MongoDB

- Vous devez installer MongoDB en fonction de votre matériel. Privilégier une solution avec Docker.
- Vous pouvez utiliser une solution graphique comme Studio3T (ou autre) ou utiliser l'interface texte en ligne de commande

## 8. Import et accès

- Télécharger le fichier gym.zip : <https://gofile.me/6Zy9q/LMiP9qFLR>
- L'importer avec la commande mongorestore :

```
mongorestore -d gym "chemin/fichiers_gym_dezippés"
```

- Une fois connecté au shell (ex. sous mac mongosh gym), il est possible de connaître l'ensemble des bases de données présentes sur le serveur. Pour ceci, vous devez utiliser la commande `show dbs`. Pour choisir la base sur laquelle vous voulez travailler, il faut la sélectionner à l'aide de la commande `use db` (dans notre cas `use gym`)

Une base de données est constituée d'une ou plusieurs collections. Chacune de celles-ci contient un ensemble de documents. Pour lister celles-ci, on utilise la commande `show collections`.

```
show collections
```

Vous devriez avoir une liste à deux éléments : Gymnases et Sportifs.

Dans MongoDB, comme nous le verrons par la suite, nous utilisons un formalisme de type `db.collection.fonction()` :

- `db` représente la base de données choisie grâce à la commande `use` (ce mot clé est non modifiable)

- collection représente la collection dans laquelle nous allons effectuer l'opération, et doit donc correspondre à une des collections présentes dans la base
- fonction() détermine l'opération à effectuer sur la collection.

En premier lieu, on peut dénombrer le nombre de documents de chaque collection, grâce à la fonction count().

```
db.Gymnases.count()  
db.Sportifs.count()
```

Les documents présents dans une collection n'ont pas de schémas prédéfinis. Si nous souhaitons avoir une idée de ce que contient la collection, il est possible d'afficher un document (le premier trouvé), avec findOne(). Cette opération permet de comprendre la structure globale d'un document, même s'il peut y avoir des différences entre documents.

```
db.Gymnases.findOne()  
db.Sportifs.findOne()
```

Il est possible d'inclure des critères de sélection dans cette fonction, que nous verrons dans la suite. De même pour la sélection des items à afficher.

Une autre fonction très utile pour mieux appréhender les données est de lister les valeurs prises par les différents items de la collection, grâce à distinct(). Pour spécifier un sous-item d'un item, il est nécessaire d'utiliser le formalisme item.sousitem.

```
db.Gymnases.distinct("Ville")  
db.Gymnases.distinct("Surface")  
db.Gymnases.distinct("Seances.Libelle")  
db.Gymnases.distinct("Seances.Jour")  
db.Sportifs.distinct("Sexe")  
db.Sportifs.distinct("Sports.Jouer")
```

## 9. Recherches d'informations

Pour faire des recherches, il existe la fonction find(). Sans paramètre, elle renvoie l'ensemble des documents. Il faut donc l'utiliser avec précautions. Mais celle-ci peut aussi prendre deux paramètres :

- les critères de sélection des documents
- les choix d'items des documents à afficher

Ces deux paramètres doivent être écrits sous la forme d'objets JSON.

Dans ce premier exemple, on cherche le (ou les) conseiller s'appelant "KERVADEC".

```
db.Sportifs.find({ "Nom": "KERVADEC" })
```

L’affichage rendu par `find()` est compact et peu lisible directement. On peut *aérer* ce rendu en ajoutant la fonction `pretty()` pour avoir une présentation propre.

```
db.Sportifs.find({ "Nom": "KERVADEC" }).pretty()
```

Si l’on désire n’afficher que certains éléments, il est possible d’ajouter un deuxième argument spécifiant les items que l’on veut (avec 1) ou qu’on ne veut pas (avec 0).

```
db.Sportifs.find({ "Nom": "KERVADEC" }, { "Nom": 1 })
```

Par défaut, l’identifiant du document, toujours nommé `_id`, est renvoyé. Pour ne pas l’avoir, il faut ainsi le préciser avec `"_id": 0`.

```
db.Sportifs.find({ "Nom": "KERVADEC" }, { "_id": 0, "Nom": 1 })
```

JavaScript étant un langage comme un autre, il est possible de définir une variable stockant les choix d’affichage en sortie, comme ci-dessous. Cela pourra permettre d’avoir un code plus lisible.

```
sortie = { "_id": 0, "Nom": 1 }  
db.Sportifs.find({ "Nom": "KERVADEC" }, sortie)
```

Le test d’égalité est aussi réalisable avec une variable numérique, comme ici où on cherche les sportifs de 32 ans.

```
db.Sportifs.find({ "Age": 32 }, sortie)
```

Pour les comparaisons, nous disposons des opérateurs `$eq` (*equal*), `$gt` (*greater than*), `$gte` (*greater than or equal*), `$lt` (*less than*), `$lte` (*less than or equal*) et `$ne` (*not equal*). Voici un exemple d’utilisation pour la recherche de sportifs de plus de 32 ans.

```
db.Sportifs.find({ "Age": { "$gte": 32 } }, { "_id": 0, "Nom": 1, "Age": 1 })
```

En plus de ces comparaisons simples, nous disposons d’opérateurs de comparaisons à une liste : `$in` (présent dans la liste) et `$nin` (non présent dans la liste). Ici, nous cherchons les sportifs qui ont soit 32 ans, soit 40 ans.

```
db.Sportifs.find({ "Age": { "$in": [ 32, 40 ] } }, { "_id": 0, "Nom": 1, "Age": 1 })
```

Les documents peuvent être complexes (c’est même le but), et les critères portent donc souvent sur des sous-items. Il faut utiliser le même formalisme que précédemment (`item.sousitem`). Il faut noter qu’on peut aller aussi loin que nécessaire dans l’utilisation du `..`. Voici donc les sportifs jouant au Basket.

```
db.Sportifs.find({ "Sports.Jouer" : "Basket ball" }, sortie)
```

Ce formalisme est le même pour indiquer un sous-item à afficher. Nous ajoutons à la requête précédent l’affichage des sports joués. Nous voyons que le résultat inclu tous les sports joués par le sportif.

```
db.Sportifs.find({ "Sports.Jouer" : "Basket ball" }, { "_id": 0, "Nom": 1, "Sports.Jouer": 1 })
```

Par défaut, si on ajoute des critères de restriction dans le premier paramètre, la recherche se fait avec un *ET* entre les critères. Nous cherchons ici les joueuses de Basket.

```
db.Sportifs.find({ "Sports.Jouer" : "Basket ball", "Sexe" : "F" }, sortie)
```

Mais si on veut faire des combinaisons autres, il existe des opérateurs logiques : *\$and*, *\$or* et *\$nor*. Ces trois opérations prennent un tableau de critères comme valeur. Nous cherchons ci-dessous les sportifs soit ayant au moins 32 ans, soit de sexe féminin.

```
db.Sportifs.find({ "$or": [ { "Age" : { "$gte" : 32 } }, { "Sexe": "F" } ] }, { "_id" : 0, "Nom" : 1, "Age" : 1, "Sexe" : 1 })
```

Comme précédemment indiqué, il est courant qu'un document ne contienne pas tous les items possibles. Si l'on cherche à tester la présence ou non d'un item, on utilise l'opérateur *\$exists* (avec *true* si on teste la présence, et *false* l'absence). Dans l'exemple qui suit, nous cherchons les sportifs qui arbitrent un sport.

```
db.Sportifs.find({ "Sports.Arbitrer" : { "$exists" : true } }, sortie)
```

Il est souvent nécessaire de faire des dénombrements suite à des sélections, pour faire des vérifications de code ou des estimations de charge (ou autre). La fonction *count()* peut ainsi s'ajouter à la suite d'une fonction *find()* pour connaître la taille du résultat. Nous cherchons ici le nombre de sportifs de sexe féminin dans la base.

```
db.Sportifs.find({ "Sexe" : "F" }).count()
```

On peut aussi limiter le nombre de documents renvoyés par la fonction *find()* en lui ajoutant la fonction *limit()*, comme ici où nous nous restreignons aux 5 premiers résultats.

```
db.Sportifs.find({ "Sexe" : "F" }).limit(5)
```

Une autre opération classique est le tri des résultats, réalisable avec la fonction *sort()*. On doit indiquer les items de tri et leur attribuer une valeur de 1 pour un tri ascendant et une valeur de -1 pour un tri descendant. On affiche ici les sportifs d'au moins 32 ans dans l'ordre croissant de leur âge.

```
db.Sportifs.find({ "Age": { "$gte": 32 } }, { "_id": 0, "Nom": 1, "Age": 1 }).sort({ "Age" : 1 })
```

Idem que précédemment, mais dans l'ordre décroissant.

```
db.Sportifs.find({ "Age": { "$gte": 32 } }, { "_id": 0, "Nom": 1, "Age": 1 }).sort({ "Age" : -1 })
```

Bien évidemment, il est possible de mettre plusieurs critères de tri, comme ci-dessous avec un tri par âge décroissant, et un tri alphabétique des noms (pour les sportifs de même âge).

```
db.Sportifs.find({ "Age": { "$gte": 32 } }, { "_id": 0, "Nom": 1, "Age": 1 }).sort({ "Age" : -1, "Nom": 1 })
```

Enfin, il est possible d'écrire les commandes sur plusieurs lignes, avec des indentations, afin de rendre les commandes plus lisibles, tel que ci-dessous.

```
db.Sportifs.find(  
    { "Age": { "$gte": 32 } },  
    { "_id": 0, "Nom": 1, "Age": 1 }  
)  
.sort(  
    { "Age" : 1, "Nom": 1 }  
)
```

## 10. Aggrégation

En plus des recherches classiques d'informations, le calcul d'agrégat est très utilisé, pour l'analyse, la modélisation ou la visualisation de données. Ce calcul s'effectue avec la fonction `aggregate()`. Celle-ci prend en paramètre un tableau d'opérations (appelé aussi pipeline), pouvant contenir les éléments suivants :

- `$project` : redéfinition des documents (si nécessaire)
- `$match` : restriction sur les documents à utiliser
- `$group` : regroupements et calculs à effectuer
- `$sort` : tri sur les agrégats
- `$unwind` : découpage de tableaux
- ...

Voici un premier exemple permettant un calcul pour toute la base. Ici, nous réalisons un dénombrement (il fait la somme de la valeur 1 pour chaque document).

```
db.Gymnases.aggregate([  
    { $group: { "_id": null, "nb": { $sum: 1 } } }  
])
```

Bien évidemment, les calculs peuvent être plus complexes. Par exemple, nous cherchons ici la surface moyenne des gymnases.

```
db.Gymnases.aggregate([  
    { $group: { "_id": null, "surfmoy": { $avg: "$Surface" } } }  
])
```



Bien évidemment, ces calculs d'agrégats peuvent se faire aussi en ajoutant des critères de regroupement. Ici, nous cherchons le nombre de gymnases par ville.

```
db.Gymnases.aggregate([
  { $group: { "_id": "$Ville", "nb": { $sum: 1 }}}
])
```

Ce résultat peut être trié en ajoutant l'action \$sort dans le tableau, avec le même mécanisme que précédemment (1 : ascendant, -1 : descendant).

```
db.Gymnases.aggregate([
  { $group: { "_id": "$Ville", "nb": { $sum: 1 }}},
  { $sort: { "nb": -1 }}
])
```

Comme vu précédemment, on peut faire tous les calculs d'agrégats classiques, comme ici avec la somme (\$sum), la moyenne (\$avg), le minimum (\$min) et le maximum (\$max).

```
db.Gymnases.aggregate([
  { $group: {
    "_id": "$Ville",
    "nb": { $sum: 1 },
    "surfaceTotale": { $sum: "$Surface" },
    "surfaceMoyenne": { $avg: "$Surface" },
    "surfaceMinimum": { $min: "$Surface" },
    "surfaceMaximum": { $max: "$Surface" }
  }}
])
```

On peut aussi faire une restriction avant le calcul, avec l'opération \$match. Ici, nous nous restreignons aux gymnases dans lesquels il y a (au moins) une séance de Volley.

```
db.Gymnases.aggregate([
  { $match: { "Seances.Libelle" : "Volley ball" }},
  { $group: { "_id": "$Ville", "nb": { $sum: 1 }}}
])
```

Imaginons maintenant que nous souhaitons calculer le nombre de séances journalières. La première idée serait de réaliser l'opération suivante.

```
db.Gymnases.aggregate([
```

```
{ $group: { "_id": "$Seances.Jour", "nb": { $sum: 1 } } }  
])
```

Malheureusement, nous voyons que le regroupement se fait par couple de jours existant dans la base. Il faut donc faire un découpage des tableaux Seances dans chaque document. Pour cela, il existe l'opération \$unwind.

Pour montrer comment fonctionne cette opération, voici les 5 premières lignes renvoyées lorsqu'on l'applique directement sur les données. On s'aperçoit que chaque document ne contient plus qu'une seule séance.

```
db.Gymnases.aggregate( [  
  { $unwind: "$Seances" },  
  { $limit: 5 }  
]).pretty()
```

Par ce biais, nous pouvons donc maintenant faire l'opération de regroupement par jour de la semaine.

```
db.Gymnases.aggregate( [  
  { $unwind: "$Seances" },  
  { $group: { "_id": "$Seances.Jour", "nb": { $sum: 1 } } }  
])
```

Nous remarquons qu'il y a des différences dans la casse des jours de la semaine. Pour gérer cela, nous transformons les jours de la semaine en *minuscule* avec la fonction toLower.

```
db.Gymnases.aggregate( [  
  { $unwind: "$Seances" },  
  { $group: { "_id": { $toLower: "$Seances.Jour" }, "nb": { $sum: 1 } } }  
])
```

Une autre façon de faire est de redéfinir les documents, pour ne garder que le jour de la semaine (en minuscule), dans celui-ci.

```
db.Gymnases.aggregate( [  
  { $unwind: "$Seances" },  
  { $project: { "Jour": { $toLower: "$Seances.Jour" } } }  
])
```

Avec ce code, il est facilement possible de passer à un décompte par jour de la semaine, comme ci-dessous.

```
db.Gymnases.aggregate( [  
  { $unwind: "$Seances" },
```

```
{ $project: { "Jour": { $toLower: "$Seances.Jour" } } },  
{ $group: { "_id": "$Jour", "nb": { $sum: 1 } } }
```

1)

## Le paradigme MapReduce en MongoDB

- MapReduce est un des mécanismes les plus complexes de requêtes de MongoDB. Il se base sur la spécification des deux fonctions Map et Reduce (écrites en javascript). Ces deux fonctions sont des fonctions définies par l'utilisateur.
- En MongoDB, la fonction Map, permet de générer des documents clés-valeurs pour les transmettre en entrée à Reduce. La fonction ne prend aucun paramètre, on accède à l'objet analysé via l'opérateur this. La fonction peut émettre des couples via la fonction emit(key, value) autant de fois que nécessaire dans la fonction.
- La fonction Reduce retourne le résultat agrégé à partir de ces documents en entrée. La fonction prend deux paramètres key et values (tableau des valeurs de la clé). Elle peut être appelée plusieurs fois pour la même clé, elle doit donc renvoyer une valeur de même type que celles dans le tableau.
- On doit passer un troisième paramètre, un littéral JSON, qui représente les options de la fonction. La principale option (out) est la collection dans laquelle le résultat sera placé. Si l'on veut voir le résultat, sans le stocker, il est possible d'indiquer out: { inline: 1 }

## 11. Répondre aux questions suivantes

1. Quels sont les sportifs (identifiant, nom et prénom) qui ont entre 20 et 30 ans ?
2. Quels sont les gymnases de "Villetaneuse" ou de "Sarcelles" qui ont une surface de plus de 400 m2 ?
3. Quels sont les sportifs (identifiant et nom) qui pratiquent du hand Ball ?
4. Dans quels gymnases et quels jours y a-t-il des séances de hand Ball ?
5. Quels sportifs (identifiant et nom) ne pratiquent aucun sport ?
6. Quels gymnases n'ont pas de séances le dimanche ?
7. Quels gymnases ne proposent que des séances de basket ball ou de volley ball ?
8. Quels sont les entraîneurs qui sont aussi joueurs ?
9. Quels sont les sportifs qui sont des conseillers ?
10. Pour le sportif "Kervadec" quel est le nom de son conseiller ?
11. Quels entraîneurs entraînent du hand ball et du basket ball ?
12. Quels sont les couples de sportifs (identifiant et nom et prénom de chaque) de même âge ?
13. Quelle est la moyenne d'âge des sportives qui pratiquent du basket ball ?
14. Quels sont les sportifs les plus jeunes ?
15. Quels entraîneurs n'entraînent que du hand ball ou du basket ball ?
16. Pour chaque sportif donner le nombre de sports qu'il arbitre
17. Pour chaque gymnase de Stains donner par jour d'ouverture les horaires des premières et dernières séances
18. Pour chaque entraîneurs de hand ball quel est le nombre de séances journalières qu'il assure ?
19. Calculer le nombre de gymnases pour chaque ville
20. Calculer le nombre de séances pour chaque jour de la semaine.
21. Calculer la superficie moyenne des gymnases pour chaque ville.