

INTRODUCTION À MONGODB

Sommaire

Définition	1
SQL vs NoSQL	1
« Tables SQL » vs « Documents NoSQL »	1
« Schéma SQL » VS « Logique NoSQL »	1
« Normalization SQL » VS « Dénormalisation NoSQL »	2
« Jointure SQL » VS « Pas de jointures dans NoSQL »	2
« Intégrité SQL » VS « NoSQL Data Integrity »	2
« CRUD SQL » VS « CRUD NoSQL »	2
MongoDB : Présentation et installation	3
Utilisation de MongoDB Client	4
Le format JSON	4
Use : Sélectionner une base de données	5
Insert() : Insérer des données dans MongoDB	6
Find() : Lire des données dans MongoDB	7
Find() : Les critères de recherches avancés	9
Find() : Filtrer les résultats	10
Find() : Trier les résultats	11
Update() : Mettre à jour des données dans MongoDB	12
Update() : Ajout & Modification d'une propriété	12
Update() : Mettre à jour plusieurs enregistrements	13
Update() : Suppression d'une propriété	13
Remove() : Supprimer un document d'une collection	14
Historique du document	16
Crédits	16

Définition

Le sigle **SQL** signifie « Structured Query Language » (Langage de requêtes structurées).

Le sigle NoSQL signifie « Not Only SQL ».

SQL vs NoSQL

Les bases de données SQL existent depuis plusieurs décennies et sont largement utilisées depuis les années 1990. Les bases de données NoSQL, même si elles existent depuis les années 60, n'ont pris de l'ampleur que récemment.

Avant de présenter la solution NoSQL « MongoDB », il est important de noter les avantages, inconvénients... bref les différences entre les bases de données relationnelles et les bases de données NoSQL.

« Tables SQL » vs « Documents NoSQL »

Les données stockées dans une base SQL sont organisées en tables reliées entre elles. La structure et le type des données sont rigides (fixées à l'avance).

Dans les projets liés à une base SQL, la base de données doit être modélisée avant d'implémenter une logique métier. Avec ce fonctionnement, il est difficile de faire des erreurs.

Dans une base NoSQL, les données sont enregistrées sous forme de « documents » eux-mêmes stockés dans des « collections ».

Là où NoSQL repousse les limites du SQL ; c'est que l'on peut stocker les données que l'on souhaite dans n'importe quel document. La base NoSQL n'a pas de structure définie à l'avance. NoSQL est plus flexible mais offre la possibilité de stocker des données n'importe où, ce qui peut entraîner des problèmes de cohérence.

« Schéma SQL » VS « Logique NoSQL »

Dans une base de données SQL, il est impossible d'ajouter des données tant que les tables ne sont pas définies (avec leurs colonnes et leur type de données). La définition des tables est ce qu'on appelle un « Schéma ». Ce schéma contient donc les informations sur la structure des tables, les clés primaires, les index, les contraintes, les déclencheurs et les procédures stockées.

Le schéma de la base de données doit être conçu et mis en œuvre avant que toute logique métier puisse être développée pour manipuler les données.

Il est évidemment possible de modifier, par la suite, un schéma SQL existant mais de gros changements rendent la mise à jour des objets métiers compliquée.

Dans une base de données NoSQL, la logique est totalement différente. Il est en effet possible d'ajouter des données n'importe où, à tout moment, sans qu'il soit nécessaire de spécifier une conception ou une collection à l'avance. Les « collections » et les « documents » sont créés à la volée (agir sur une collection inexistante la créera automatiquement).

Les bases de données NoSQL sont probablement plus adaptées aux projets où les exigences initiales en matière de données sont difficiles à déterminer. Attention toutefois à ne pas confondre « difficulté » et « paresse ».

Négliger la conception d'un bon stock de données peut drastiquement réduire vos chances de succès.

« Normalization SQL » VS « Dénormalisation NoSQL »

Le principe de « Normalisation » et de « Dénormalisation », précise la façon dont les données sont dupliquées (NoSQL) ou reliées par des clés étrangères (SQL).

La normalisation SQL réduit au minimum la redondance des données et assure les liens entre les tables grâce à des références entre elles (clés étrangères).

Le principe de dénormalisation admet une duplication des données ce qui conduit à des requêtes beaucoup plus rapides en lecture mais en contrepartie, beaucoup plus lentes en écriture (si l'écriture concerne plusieurs enregistrements).

« Jointure SQL » VS « Pas de jointures dans NoSQL »

Les requêtes SQL permettent l'utilisation d'une clause JOIN permettant d'obtenir des données de plusieurs tables en une seule instruction SQL.

NoSQL n'implémente pas toujours cette clause. Par exemple, dans MongoDB, il sera nécessaire de récupérer séparément les différents éléments et de les associer en implémentant la logique métier adaptée.

« Intégrité SQL » VS « NoSQL Data Integrity »

La plupart des moteurs SQL permettent d'appliquer des règles d'intégrité de données à l'aide de contraintes de clés étrangères. Cela empêche notamment de supprimer un élément référencé dans d'autres éléments.

Le schéma SQL applique ces règles pour prévenir la création de données invalides ou d'enregistrements orphelins.

Ces options n'existent pas dans les moteurs NoSQL. Il est possible de stocker ce que l'on veut indépendamment de tout autre document issu de la même collection. Pour « simuler » ces options d'intégrité, il faudra, une fois de plus, implémenter la logique dans les classes métier.

« CRUD SQL » VS « CRUD NoSQL »

SQL est un langage déclaratif devenu depuis une norme internationale.

Le CRUD SQL contient généralement : CREATE, SELECT, UPDATE, DELETE.

Le CRUD NoSQL ressemblerait plus à INSERT, FIND, UPDATE, REMOVE (cela peut varier d'un moteur NoSQL à l'autre).

Malgré les différences dans les termes utilisés, l'objectif reste le même : Lire, insérer, mettre à jour et supprimer des données.

MongoDB : Présentation et installation

MongoDB est un système de gestion de bases de données de la mouvance « NoSQL ». Il est orienté « document » et sa philosophie est de pouvoir gérer de (très) grandes quantités de données. Cette philosophie permet à MongoDB d'être idéal dans un contexte « [Big Data](#) ».

Installation de MongoDB

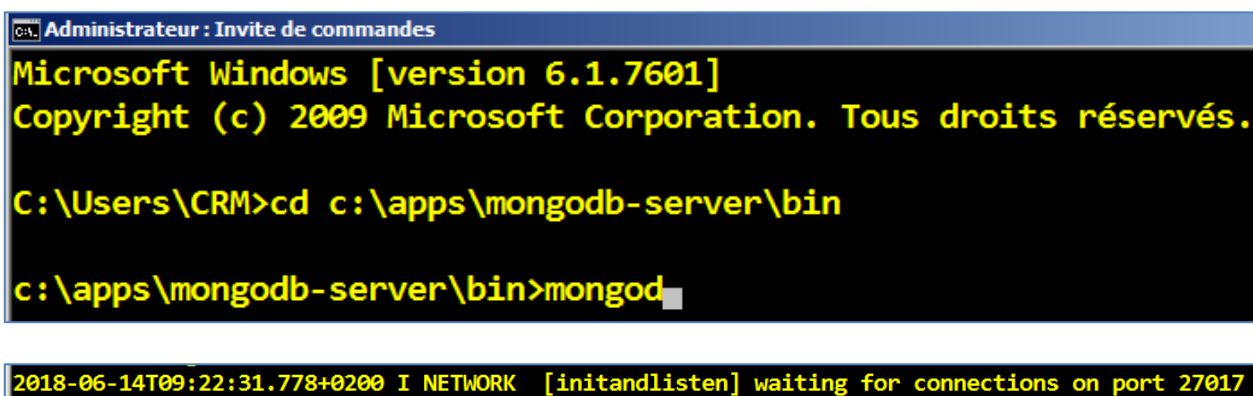
/ ! \ MongoDB est peut-être déjà installé sur votre machine ! Si tel est le cas, passez directement à l'étape suivante.

Se rendre sur le site officiel de MongoDB : <http://www.mongodb.org/> puis récupérer la dernière version stable du logiciel

Une fois installé, vérifier la présence du répertoire « **C:\data\db** ». S'il n'existe pas, le créer. Par défaut, MongoDB stocke les bases de données dans ce répertoire.

Lancement de MongoDB Server

Ouvrir une invite de commande, naviguer jusqu'au dossier « bin » se trouvant dans le répertoire d'installation de MongoDB puis lancer l'exécutable « mongod ». Si le démarrage est réussi, un message du type « Waiting for connections on port 27017 » apparaît à la fin du processus de démarrage.



```
Administrateur : Invite de commandes
Microsoft Windows [version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Tous droits réservés.

C:\Users\CRM>cd c:\apps\mongodb-server\bin

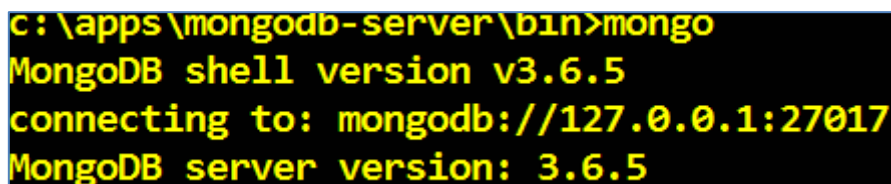
c:\apps\mongodb-server\bin>mongod

2018-06-14T09:22:31.778+0200 I NETWORK [initandlisten] waiting for connections on port 27017
```

Tant que la fenêtre du terminal est ouverte, le serveur est en fonctionnement et à l'écoute. Fermer la fenêtre met fin au processus et stoppe le serveur MongoDB.

Lancement de MongoDB Client

Tout en laissant la fenêtre du serveur MongoDB ouverte, ouvrir une seconde invite de commande puis lancer l'exécutable « mongo ». Vous êtes désormais connecté avec le moteur de base de données MongoDB.



```
c:\apps\mongodb-server\bin>mongo
MongoDB shell version v3.6.5
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.6.5
```

Dans certains cas, le terminal peut afficher quelques avertissements, ignorez les pour le moment.

Utilisation de MongoDB Client

Là où les moteurs SQL stockent les données en ligne dans des tables, MongoDB modélise les informations sous forme de documents au format JSON (Javascript Object Notation) et les stocke en au format BSON (Le format BSON est en réalité du JSON au format binaire).

Plus concrètement, on envoie des objets JSON dans les requêtes d'écriture, MongoDB les stocke au format BSON. Lorsque nous récupérons un jeu de résultat à partir d'une requête en lecture, des objets au format JSON sont retournés et directement exploitables par de nombreux langages tels que Javascript, PHP, C#, et bien d'autres...

De plus, les paramètres des différentes méthodes de MongoDB sont au format JSON.

Le format JSON

Le format [JSON](#) se base sur des paires « Clé/valeur » où la valeur peut être :

- 1) Une chaîne de caractères
- 2) Un type numérique (entier, flottant...)
- 3) Un tableau
- 4) Un autre objet JSON
- 5) La valeur « null »

Exemple :

```
{
  "nom": "DEVOLDERE",
  "prenom": "Mickaël",
  "surnom": null,
  "age": 37,
  "interets": ["AstroBiology", "Networks", "Web Development"],
  "projets": [
    { "nomProjet": "NetworkSpeed", "version": "1.1.0"},
    { "nomProjet": "RegexTester", "version": "2.3.7"}
  ]
}
```

- Les accolades de début et fin encapsulent l'objet JSON
- Les entrées « nom », « prénom » ont pour valeur une chaîne de caractère
- L'entrée surnom a pour valeur « null »
- L'entrée « âge » a pour valeur un entier
- L'entrée « intérêts » a pour valeur un tableau
- L'entrée « projets » a pour valeur un tableau contenant des objets JSON

Use : Sélectionner une base de données

Avant de pouvoir manipuler des données dans MongoDB, il est nécessaire de sélectionner une base de données.

Pour cela, nous utiliserons l'instruction « use », identique au langage SQL.

Dans le terminal du client MongoDB, sélectionnez la base de données « people »

```
> use people  
switched to db people
```

MongoDB vous indique qu'il a basculé vers la base de données sélectionnée.

Sélectionner la base ? Mais nous n'en avons créé aucune !

Exact, si la base de données demandée n'existe pas, MongoDB la créera automatiquement.

Collections et Documents

Dans une base de données relationnelle (SQL) on crée des tables pour y stocker nos données.

Dans MongoDB, une base de données contient des « [collections](#) » dans lesquelles on ajoute des « [documents](#) ». Une collection est donc un ensemble de documents de même nature.

Tout comme pour les bases de données, les collections sont créées implicitement dès qu'un document y est inséré.

MongoDB limite la taille des documents à 16 Mo (ici, un utilisateur = un document). Si une collection contient des documents qui approchent cette limite, il peut être judicieux de la redécouper en plusieurs collections.

Insert() : Insérer des données dans MongoDB

Nous allons créer une collection d'**utilisateurs**. La création de la collection est implicite, elle se fait automatiquement à l'insertion du premier document.

Pour insérer un premier document nous utiliserons la commande Insert() qui accepte 1 paramètre : l'objet JSON à insérer.

Insertion d'un nouvel utilisateur dans notre base de données « people » :

```
db.utilisateurs.insert({nom:"DEVOLDERE", prenom:"Mickaël"})
```

Si elle n'existe pas, la collection "utilisateurs" est automatiquement créée dans la base de données en cours d'utilisation

Insérons maintenant une seconde personne :

```
db.utilisateurs.insert({nom:"CHATELOT", prenom:"Franck"})
```

Une fois les 2 commandes saisies et validées, le terminal devrait ressembler à ceci :

```
> db.utilisateurs.insert({nom:"DEVOLDERE", prenom:"Mickaël"})
WriteResult({ "nInserted" : 1 })
> db.utilisateurs.insert({nom:"CHATELOT", prenom:"Franck"})
WriteResult({ "nInserted" : 1 })
```

Pour chaque insertion, MongoDB vous retourne le nombre de lignes ajoutées, ce qui confirme que l'insertion s'est bien déroulée.

Find() : Lire des données dans MongoDB

Nous allons maintenant pouvoir vérifier si nos données ont bien été inscrites dans la base de données.

Pour cela, nous utiliserons la commande « find() »

```
db.utilisateurs.find()
```

La commande devrait vous retourner les 2 personnes précédemment créées :

```
> db.utilisateurs.find()
{ "_id" : ObjectId("5b2221c5e67a6b27e32bbab8"), "nom" : "DEVOLDERE", "prenom" : "Mickaël" }
{ "_id" : ObjectId("5b2221d3e67a6b27e32bbab9"), "nom" : "CHATELOT", "prenom" : "Franck" }
>
```

Remarquez la présence de la propriété "_id" qui est un identifiant unique pour le document.

S'il n'est pas spécifié à l'insertion, MongoDB génère un unique [ObjectId](#) qui identifie le document.

Cet identifiant sera utilisé pour créer des liens entre 2 collections, un peu comme les clés étrangères en SQL.

Cependant, MongoDB ne permet pas les jointures dans les requêtes de lecture. Il faudra donc exécuter plusieurs requêtes pour récupérer tous les objets liés.

MongoDB est « **schemaless** », c'est à dire qu'aucun schéma n'est défini à l'avance et que les documents peuvent ne pas respecter le même format (attention, trop de disparités dans une même collection peut être source de bugs si elles sont mal gérées coté logiciel).

Par exemple, si nous ajoutons un nouvel utilisateur possédant un attribut supplémentaire, aucun problème !

```
> use people
switched to db people
> db.utilisateurs.find()
{ "_id" : ObjectId("5b2221c5e67a6b27e32bbab8"), "nom" : "DEVOLDERE", "prenom" : "Mickaël" }
{ "_id" : ObjectId("5b2221d3e67a6b27e32bbab9"), "nom" : "CHATELOT", "prenom" : "Franck" }
>
> db.utilisateurs.insert({nom:"Golay", prenom:"Jerry", Origine: "Pluton"})
WriteResult({ "nInserted" : 1 })
> db.utilisateurs.find()
{ "_id" : ObjectId("5b2221c5e67a6b27e32bbab8"), "nom" : "DEVOLDERE", "prenom" : "Mickaël" }
{ "_id" : ObjectId("5b2221d3e67a6b27e32bbab9"), "nom" : "CHATELOT", "prenom" : "Franck" }
{ "_id" : ObjectId("5b2a0f0cfce3684cc3e511fa"), "nom" : "Golay", "prenom" : "Jerry", "Origine" : "Pluton" }
```

Nous pouvons voir ici que l'utilisateur « Jerry Golay » possède un attribut « Origine » absent des autres entités de la même collection.

J'attire une nouvelle fois votre attention sur le fait qu'abuser de cette souplesse peut-être contre-productif.

Pour connaître le nombre de résultats retournés par une requête, nous utiliserons la méthode « count() » en complément de la méthode « find() » comme dans l'exemple suivant :

```
> db.utilisateurs.find().count()
6
```

Le résultat de cette requête tient compte des utilisateurs insérés dans la partie suivante

MongoDB renvoie un entier correspondant aux nombre de documents concernés par une requête.

Find() : Les critères de recherche

Ajoutons quelques utilisateurs à notre collection « people »

```
> db.utilisateurs.insert({nom:"Thiry", prenom:"Sophie", Origine: "Terre"})
WriteResult({ "nInserted" : 1 })
> db.utilisateurs.insert({nom:"Roche", prenom:"Didier", Origine: "Terre"})
WriteResult({ "nInserted" : 1 })
> db.utilisateurs.insert({nom:"DEVOLDERE", prenom:"Jean", Origine: "Terre"})
WriteResult({ "nInserted" : 1 })
```

Nous avons ajouté 3 utilisateurs ayant pour propriété commune « Origine ». La collection contient désormais 6 utilisateurs.

```
> db.utilisateurs.find()
{ "_id" : ObjectId("5b2221c5e67a6b27e32bbab8"), "nom" : "DEVOLDERE", "prenom" : "Mickaël" }
{ "_id" : ObjectId("5b2221d3e67a6b27e32bbab9"), "nom" : "CHATELOT", "prenom" : "Franck" }
{ "_id" : ObjectId("5b2a0f0cfce3684cc3e511fa"), "nom" : "Golay", "prenom" : "Jerry", "Origine" : "Pluton" }
{ "_id" : ObjectId("5bb5ef189a2078184e450f06"), "nom" : "Thiry", "prenom" : "Sophie", "Origine" : "Terre" }
{ "_id" : ObjectId("5bb5ef389a2078184e450f07"), "nom" : "Roche", "prenom" : "Didier", "Origine" : "Terre" }
{ "_id" : ObjectId("5bb5ef5a9a2078184e450f08"), "nom" : "DEVOLDERE", "prenom" : "Jean", "Origine" : "Terre" }
```

Une collection pouvant être de taille conséquente, il est possible de n'en récupérer qu'une partie.

La commande « find() » accepte 2 paramètres optionnels :

- 1^{er} paramètre : Un objet JSON qui permet de spécifier une ou plusieurs conditions (critère de recherche équivalent de la clause WHERE en langage SQL).
- 2nd paramètre : Un objet JSON qui permet de filtrer les propriétés que l'on souhaite récupérer (équivalent de la sélection de colonnes en langage SQL).

Exemple : Récupérer tous les utilisateurs dont la valeur de la propriété « Origine » est égale à « Terre » :

```
> db.utilisateurs.find({Origine: "Terre"})
{ "_id" : ObjectId("5bb5ef189a2078184e450f06"), "nom" : "Thiry", "prenom" : "Sophie", "Origine" : "Terre" }
{ "_id" : ObjectId("5bb5ef389a2078184e450f07"), "nom" : "Roche", "prenom" : "Didier", "Origine" : "Terre" }
{ "_id" : ObjectId("5bb5ef5a9a2078184e450f08"), "nom" : "DEVOLDERE", "prenom" : "Jean", "Origine" : "Terre" }
```

Nous remarquons que tous les utilisateurs concernés sont récupérés.

Il est possible de spécifier plusieurs conditions dans la même commande :

```
> db.utilisateurs.find({nom: "DEVOLDERE", Origine: "Terre"})
{ "_id" : ObjectId("5bb5ef5a9a2078184e450f08"), "nom" : "DEVOLDERE", "prenom" : "Jean", "Origine" : "Terre" }
```

Remarquez que le document correspondant à « DEVOLDERE Mickaël » n'a pas été récupéré car il ne possède pas la propriété « Origine ».

Find() : Les critères de recherches avancés

Dans les exemples précédents, nous récupérons des documents selon une ou plusieurs conditions d'égalité.

Tout comme SQL, il est possible d'affiner les critères de recherche grâce à des [opérateurs](#) implémentés dans MongoDB dont voici les principaux :

Opérateur	Signification
\$eq	Est égal à
\$ne	N'est pas égal à
\$gt	Supérieur à
\$lt	Inférieur à
\$gte	Supérieur ou égal à
\$lte	Inférieur ou égal à
\$or	« Ou » logique
\$and	« Et » logique
\$in	Est égal à l'une des valeurs
\$nin	N'est pas égal à une des valeurs OU le champ n'existe pas
\$exists	Pour vérifier l'existence ou l'absence d'un champ dans un document

Exemples :

Note : MongoDB est sensible à la casse. Ainsi, les valeurs « pluton » et « Pluton » sont considérées comme différentes de même que les champs « origine » et « Origine » qui sont tous deux différents.

« \$eq, \$ne » : Tous les utilisateurs dont l'Origine n'est pas Pluton :

```
> db.utilisateurs.find({ Origine: {$ne: "Pluton"} })
```

« \$gt, \$gte, \$lt, \$lte » : Tous les utilisateurs ayant leur âge compris entre 18 et 99 ans.

```
> db.utilisateurs.find({ age: { $gte:18, $lt:100 } })
```

« \$and, \$or » : Tous les utilisateurs ayant pour nom « DEVOLDERE » OU pour origine « Pluton »

Note : \$and et \$or s'utilisent de la même manière.

```
> db.utilisateurs.find({ $or: [{ nom:"DEVOLDERE"}, {Origine: "Pluton"} ] })
{ "_id" : ObjectId("5b2221c5e67a6b27e32bbab8"), "nom" : "DEVOLDERE", "prenom" : "Mickaël" }
{ "_id" : ObjectId("5b2a0f0cfce3684cc3e511fa"), "nom" : "Golay", "prenom" : "Jerry", "Origine" : "Pluton" }
{ "_id" : ObjectId("5bb5ef5a9a2078184e450f08"), "nom" : "DEVOLDERE", "prenom" : "Jean", "Origine" : "Terre" }
```

Pour voir d'autres exemples, vous pouvez consulter la documentation officielle de MongoDB :

<https://docs.mongodb.com/manual/reference/operator/>

Find() : Filtrer les résultats

Pour le moment, chaque requête renvoie les documents concernés dans leur ensemble. Si les documents sont volumineux, il peut être intéressant de ne récupérer que les valeurs dont nous avons besoin.

C'est le rôle du 2nd paramètre de la méthode « find() » que l'on pourrait considérer comme l'équivalent de la sélection de colonne en SQL.

Il s'agit de spécifier les champs que l'on souhaite récupérer en indiquant leur nom et en attribuant la valeur « 1 ». Pour spécifier à MongoDB que l'on ne souhaite pas récupérer un champ en particulier, il suffit de le renseigner avec la valeur « 0 ».

Reprenons notre base de données d'utilisateurs.

Si nous exécutons la requête suivante :

```
> db.utilisateurs.find({}, {nom:1, prenom:1})
```

MongoDB renverra tous les utilisateurs (1^{er} paramètre vide = pas de critère de recherche) mais les résultats ne contiendront que les champs noms et prénoms de chaque document :

```
> db.utilisateurs.find({}, {nom:1, prenom:1})
{ "_id" : ObjectId("5b2221c5e67a6b27e32bbab8"), "nom" : "DEVOLDERE", "prenom" : "Mickaël" }
{ "_id" : ObjectId("5b2221d3e67a6b27e32bbab9"), "nom" : "CHATELOT", "prenom" : "Franck" }
{ "_id" : ObjectId("5b2a0f0cfce3684cc3e511fa"), "nom" : "Golay", "prenom" : "Jerry" }
{ "_id" : ObjectId("5bb5ef189a2078184e450f06"), "nom" : "Thiry", "prenom" : "Sophie" }
{ "_id" : ObjectId("5bb5ef389a2078184e450f07"), "nom" : "Roche", "prenom" : "Didier" }
{ "_id" : ObjectId("5bb5ef5a9a2078184e450f08"), "nom" : "DEVOLDERE", "prenom" : "Jean" }
```

Tu es bien gentil toi, mais les résultats contiennent l'attribut « _id » alors que je ne l'ai pas demandé !

C'est exact, l'identifiant d'un document est toujours retourné sauf si on précise explicitement que l'on ne souhaite pas le récupérer en précisant le nom du champ suivi de la valeur « 0 » :

```
> db.utilisateurs.find({}, {_id:0, nom:1, prenom:1})
{ "nom" : "DEVOLDERE", "prenom" : "Mickaël" }
{ "nom" : "CHATELOT", "prenom" : "Franck" }
{ "nom" : "Golay", "prenom" : "Jerry" }
{ "nom" : "Thiry", "prenom" : "Sophie" }
{ "nom" : "Roche", "prenom" : "Didier" }
{ "nom" : "DEVOLDERE", "prenom" : "Jean" }
```

Nos résultats sont maintenant correctement filtrés.

Find() : Trier les résultats

Tout comme en SQL, MongoDB permet de trier le jeu de résultat avant de le retourner.

Pour trier les résultats d'une requête, nous utiliserons la méthode « sort() ».

La méthode « sort() » prend 1 paramètre qui est un objet JSON précisant la colonne sur laquelle sera effectuée le tri ainsi que l'ordre de tri (croissant ou décroissant).

Pour illustrer cette méthode, nous allons maintenant trier nos utilisateurs selon leur nom, par ordre croissant :

```
> db.utilisateurs.find({}, {_id:0,nom:1, prenom:1}).sort({nom:1})
{ "nom" : "CHATELOT", "prenom" : "Franck" }
{ "nom" : "DEVOLDERE", "prenom" : "Mickaël" }
{ "nom" : "DEVOLDERE", "prenom" : "Jean" }
{ "nom" : "Golay", "prenom" : "Jerry" }
{ "nom" : "Roche", "prenom" : "Didier" }
{ "nom" : "Thiry", "prenom" : "Sophie" }
```

Et la même par ordre décroissant :

```
> db.utilisateurs.find({}, {_id:0,nom:1, prenom:1}).sort({nom:-1})
{ "nom" : "Thiry", "prenom" : "Sophie" }
{ "nom" : "Roche", "prenom" : "Didier" }
{ "nom" : "Golay", "prenom" : "Jerry" }
{ "nom" : "DEVOLDERE", "prenom" : "Mickaël" }
{ "nom" : "DEVOLDERE", "prenom" : "Jean" }
{ "nom" : "CHATELOT", "prenom" : "Franck" }
```

Remarquez la petite différence entre les 2 requêtes

La différence se situe au niveau de la valeur de la colonne de tri : « 1 » pour trier par ordre croissant et « -1 » pour trier par ordre décroissant.

Update() : Mettre à jour des données dans MongoDB

La commande Update permet de mettre à jour un ou plusieurs enregistrements d'une collection.

Cette commande accepte 2 paramètres au minimum et un 3^{ème} paramètre optionnel.

Comme pour la méthode « find() », le 1^{er} paramètre sert de critère de recherche.

Le 2^{ème} paramètre de la méthode « update() » va servir à définir la mise à jour à appliquer aux documents correspondant aux critères de recherches.

Un 3^{ème} paramètre facultatif servira à ajouter des options à la requête, nous y reviendrons plus tard.

L'état de la collection d'utilisateurs avant les appels à update() :

```
> db.utilisateurs.find({}, {_id:0})
{ "nom" : "DEVOLDERE", "prenom" : "Mickaël" }
{ "nom" : "CHATELOT", "prenom" : "Franck" }
{ "nom" : "Golay", "prenom" : "Jerry", "Origine" : "Pluton" }
{ "nom" : "Thiry", "prenom" : "Sophie", "Origine" : "Terre" }
{ "nom" : "Roche", "prenom" : "Didier", "Origine" : "Terre" }
{ "nom" : "DEVOLDERE", "prenom" : "Jean", "Origine" : "Terre" }
```

Update() : Ajout & Modification d'une propriété

Mettons maintenant à jour nos utilisateurs. Disons que nous souhaitons ajouter un champ « humain » ayant la valeur « true » aux utilisateurs dont l'origine est « Terre ».

Exécutons une 1^{ère} requête de mise à jour :

```
> db.utilisateurs.update( { Origine:"Terre" }, { $set: { Humain:true } } )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

Remarquez l'utilisation du mot clé « \$set » dans le second paramètre. « \$set » permet de spécifier les champs à mettre à jour. Si les champs spécifiés n'existent pas, ils sont tout simplement créés.

Si nous affichons à nouveau notre collection d'utilisateurs, que pouvez-vous remarquer ?

```
> db.utilisateurs.find({}, {_id:0})
{ "nom" : "DEVOLDERE", "prenom" : "Mickaël" }
{ "nom" : "CHATELOT", "prenom" : "Franck" }
{ "nom" : "Golay", "prenom" : "Jerry", "Origine" : "Pluton" }
{ "nom" : "Thiry", "prenom" : "Sophie", "Origine" : "Terre", "Humain" : true }
{ "nom" : "Roche", "prenom" : "Didier", "Origine" : "Terre" }
{ "nom" : "DEVOLDERE", "prenom" : "Jean", "Origine" : "Terre" }
```

Un seul enregistrement a été mis à jour. Par défaut, MongoDB ne met à jour que le 1^{er} enregistrement trouvé correspondant aux critères de recherche.

Update() : Mettre à jour plusieurs enregistrements

Pour préciser à MongoDB de mettre à jour tous les enregistrements correspondant aux critères de recherche, il faudra renseigner le 3^{ème} paramètre et attribuer la valeur « true » à un champ nommé « multi » comme dans l'exemple suivant :

```
> db.utilisateurs.update( { Origine:"Terre" }, { $set: { Humain:true } }, {multi:true} )
WriteResult({ "nMatched" : 3, "nUpserted" : 0, "nModified" : 2 })
```

Vérifions l'état de notre collection utilisateurs :

```
> db.utilisateurs.find({}, {_id:0})
{ "nom" : "DEVOLDERE", "prenom" : "Mickaël" }
{ "nom" : "CHATELOT", "prenom" : "Franck" }
{ "nom" : "Golay", "prenom" : "Jerry", "Origine" : "Pluton" }
{ "nom" : "Thiry", "prenom" : "Sophie", "Origine" : "Terre", "Humain" : true }
{ "nom" : "Roche", "prenom" : "Didier", "Origine" : "Terre", "Humain" : true }
{ "nom" : "DEVOLDERE", "prenom" : "Jean", "Origine" : "Terre", "Humain" : true }
```

Notre collection a correctement été mise à jour.

Update() : Suppression d'une propriété

Nous allons maintenant supprimer une propriété. Pour cela, nous utiliserons le mot clé « \$unset » qui s'utilise de la même manière que « \$set ».

Suppression de la propriété « Origine » si sa valeur est différente de « Terre »

```
> db.utilisateurs.update( { Origine: {$ne: "Terre"} }, { $unset: { Origine:1 } }, {multi:true} )
WriteResult({ "nMatched" : 3, "nUpserted" : 0, "nModified" : 1 })
```

Vérifions notre collection :

```
> db.utilisateurs.find({}, {_id:0})
{ "nom" : "DEVOLDERE", "prenom" : "Mickaël" }
{ "nom" : "CHATELOT", "prenom" : "Franck" }
{ "nom" : "Golay", "prenom" : "Jerry" }
{ "nom" : "Thiry", "prenom" : "Sophie", "Origine" : "Terre", "Humain" : true }
{ "nom" : "Roche", "prenom" : "Didier", "Origine" : "Terre", "Humain" : true }
{ "nom" : "DEVOLDERE", "prenom" : "Jean", "Origine" : "Terre", "Humain" : true }
```

Notre utilisateur « Jerry Golay » qui était originaire de Pluton, ne possède plus la propriété « Origine ».

Remove() : Supprimer un document d'une collection

La méthode « Remove() » permet la suppression de documents. C'est l'équivalent de l'instruction « DELETE » en langage SQL.

Cette méthode prend 1 paramètre qui servira de critère de recherche. Ce paramètre se comporte de la même manière que le 1^{er} paramètre des méthodes « find() et update() ».

Par exemple pour supprimer les utilisateurs dont la propriété « Origine » a pour valeur « Terre », nous utiliserons la commande suivante :

```
> db.utilisateurs.remove({Origine: "Terre"})
WriteResult({ "nRemoved" : 3 })
```

/! \ Contrairement à la méthode « update() » qui met à jour un seul enregistrement par défaut, la méthode « remove() » supprime TOUS les documents correspondant au critère de recherche.
C'est un détail à ne pas négliger !!!

```
> db.utilisateurs.find({}, {_id:0})
{ "nom" : "DEVOLDERE", "prenom" : "Mickaël" }
{ "nom" : "CHATELOT", "prenom" : "Franck" }
{ "nom" : "Golay", "prenom" : "Jerry" }
```

Ici, tous nos utilisateurs ayant La propriété « Origine » ayant la valeur »Terre » ont été supprimés de notre collection.





Résumé et conclusion

De plus en plus utilisé, notamment dans le « Big Data », MongoDB est l'un des moteurs NoSQL les plus utilisés

- SQL signifie « Structured Query Langage »
- NoSQL signifie « Not Only SQL »
- MongoDB est un moteur de base de données NoSQL
- MongoDB est schemaless
- MongoDB utilise le format de données JSON
- Les paramètres des fonctions MongoDB sont des objets JSON
- Les résultats retournés par MongoDB sont des objets JSON
- La commande MongoDB « find() » est l'équivalent de l'instruction SQL « SELECT FROM »
- La commande MongoDB « insert() » est l'équivalent de l'instruction SQL « INSERT INTO »
- La commande MongoDB « update() » est l'équivalent de l'instruction SQL « UPDATE SET »
- La commande MongoDB « remove() » est l'équivalent de l'instruction SQL « DELETE FROM »

--- FIN DU DOCUMENT ---

La reproduction partielle ou intégrale du présent document sur un support, quel qu'il soit, est formellement interdite sans l'accord écrit et préalable du Centre de Réadaptation de Mulhouse.

Légende des icônes	
	Information complémentaire
	Point d'attention particulier
	Intervention du formateur possible
	Lien vers une ressource externe

Historique du document

Auteur	Date	Observations
Mickaël DEVOLDÈRE	28/09/2018	Création du document

Crédits