

# MongoDB

# MongoDB

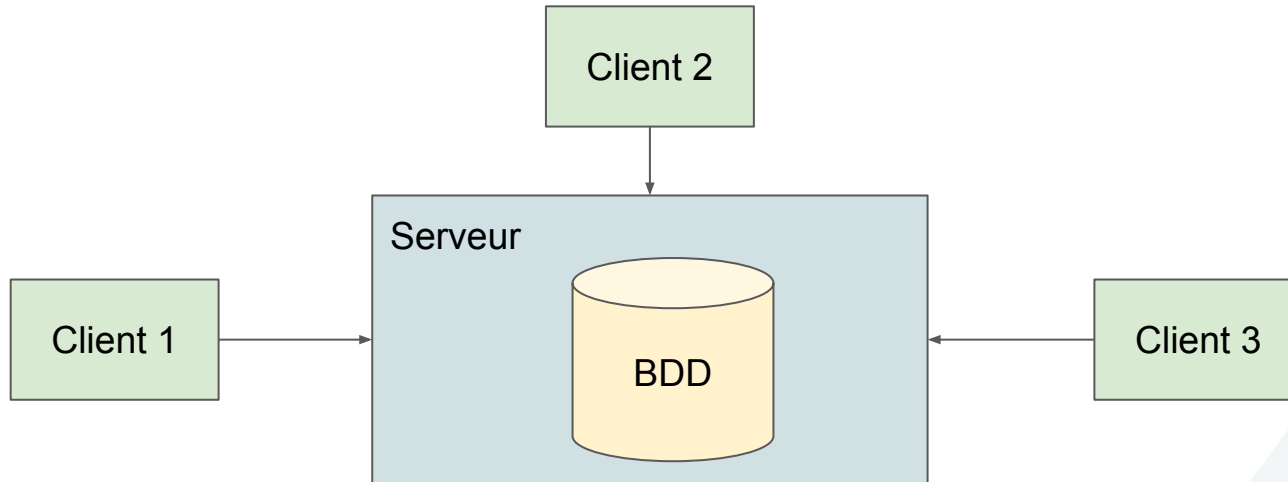
MongoDB en quelques mots :

- C'est une base de données
- MongoDB est Open Source
- MongoDB est NoSQL, donc pas de langage SQL

# Base de données

Une base de données sert à stocker de l'information.

La base de données tourne sur un serveur. Plusieurs clients peuvent simultanément lire et écrire dans la base de données.



# MongoDB

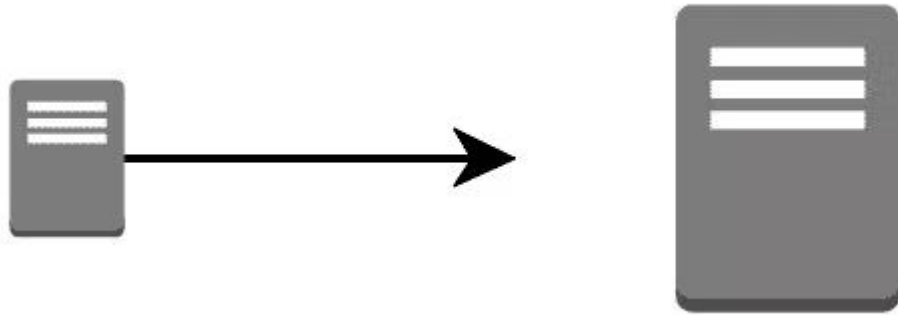
Une base de données spécialisée dans la scalabilité horizontale.

On peut ajouter de la capacité en augmentant le nombre de machines.



# MongoDB

A contrario, une base de données SQL utilise généralement une scalabilité verticale, on augmente la puissance de la machine :



# MongoDB

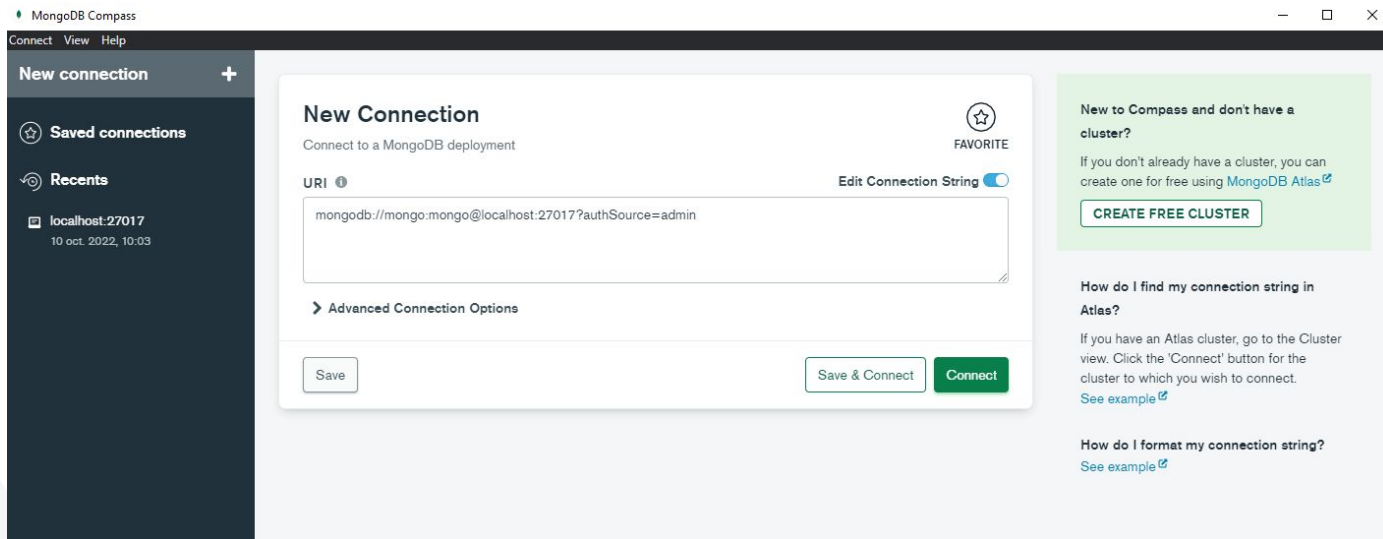
Les microservices qui séparent un serveur en plusieurs serveurs est une première forme de scalabilité horizontale, mais elle a ses limites dans certains cas extrêmes (exemple : les postes sur facebook et twitter, non divisible en microservices).



# MongoDB IDE

MongoDB Compass :

<https://www.mongodb.com/try/download/compass2>



# MongoDB IDE

MongoDB Shell:

<https://www.mongodb.com/try/download/shell2>



# MongoDB via Docker

En ligne de commandes :

// On récupère l'image

```
docker pull mongo
```

// On lance l'image

```
docker run -d --name mongoDB -p 27017:27017 -e MONGO_INITDB_ROOT_USERNAME=mongo -e  
MONGO_INITDB_ROOT_PASSWORD=mongo -v ~/mongo-data:/data/db mongo
```

# MongoDB via Docker

`docker run`

`-d` = en tâche de fond

`--name mongoDB` = le nom du container

`-p 27017:27017` = on expose le port du container sur le même port sur notre machine locale

`-e MONGO_INITDB_ROOT_USERNAME=mongo` = on initialise le nom de l'admin

`-e MONGO_INITDB_ROOT_PASSWORD=mongo` = on initialise le password de l'admin

`-v ~/mongo-data:/data/db` = répertoire où MongoDB va écrire (ici ~/mongo-data)

`mongo` = le nom de l'image

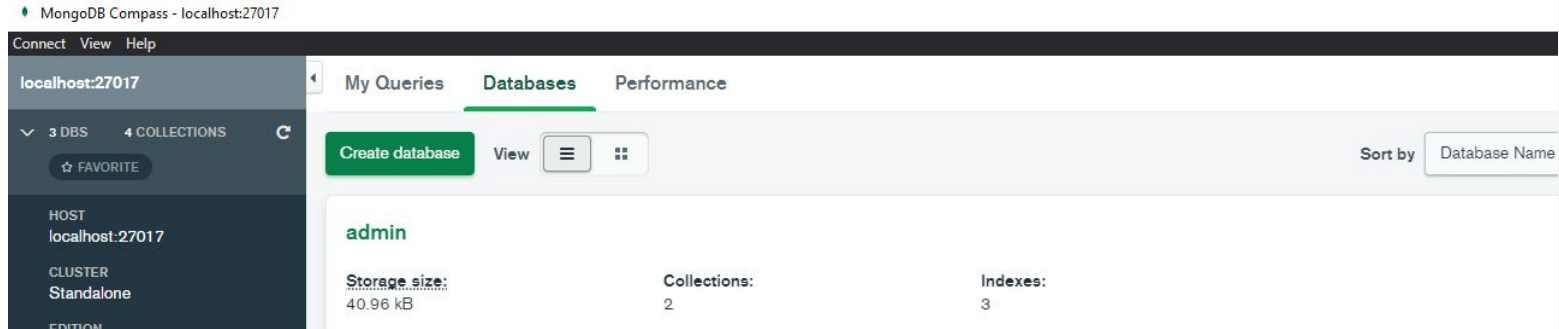
# Se connecter

Via MongoDB Compass :

```
mongodb://mongo:mongo@localhost:27017?authSource=admin
```

# Créer une base de données

Dans l'onglet Databases, cliquez sur Create database



# MongoDB

Database = la base de données de votre application

Collection = similaire à une table MySQL

Mais attention, MongoDB est NoSQL, nous allons devoir penser différemment.  
La scalabilité horizontale a un prix.

# MongoDB

MongoDB travaille via des “collections”.

Chaque collection contient des documents.

Un document est une entrée JSON au format binaire (dites BSON).

On utilise le format binaire pour des raisons d'optimisation.

Pour s'aider, on peut se dire qu'un document est une ligne dans une table MySQL (mais attention à ce genre de raccourci !).

# MongoDB

Un document est un BSON (JSON binaire). Il a la même forme et les mêmes propriétés que du JSON :

```
{
  "field1" : "value1",
  "field2" : {
    "nestedField1" : "nestedValue1",
    "nestedField2" : "nestedValue2"
  },
  "field3" : [2, 3, 4]
}
```

# MongoDB

Une “ligne” embarque directement ses dépendances en tant que BSON enfant.

Il est également possible de référencer le BSON enfant (comme une foreign key MySQL).



# MongoDB

Exemple “embedded”

```
{  
  "name" : "Toto",  
  "items" : [  
    {  
      "name" : "iron sword",  
      "damage" : 10  
    },  
  ],  
}
```

# MongoDB

Exemple “referenced”

```
{
  "name" : "Toto",
  "items" : [
    ObjectId("swordId")
  ],
}
```

```
{
  "_id" : ObjectId("swordId")
  "name" : "Iron sword",
  "damage" : 20
}
```

# MongoDB

Le design “NoSQL” se résume à :

- On fait ce que l'on veut
- On choisit de faire ce qui nous arrange le mieux pour notre application

A contrario de MySQL :

- On a créé un standard à appliquer
- Ce standard répond à toutes les situations

# MongoDB



# MongoDB

Les standards existent pour une raison !

Le NoSQL existe à cause de besoins extrêmes en termes de performance.

C'est un peu comme choisir de ne pas mettre de Getter et de laisser son attribut public pour des raisons de performances.

C'est un cas incroyable rare.

# MongoDB

Le but du jeu en MongoDB :

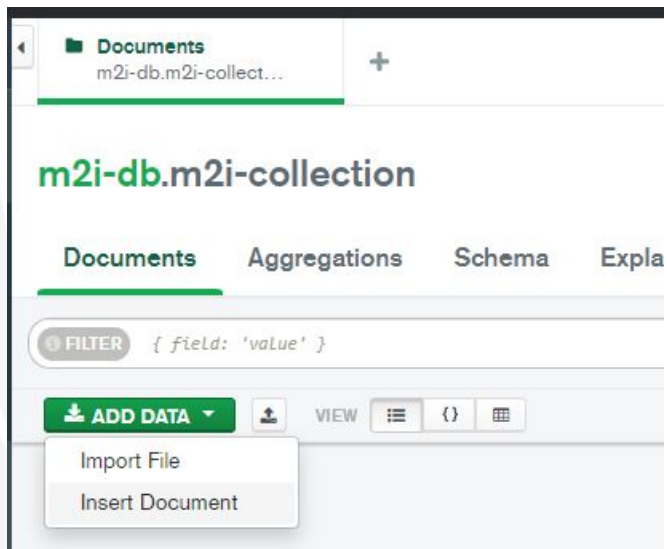
*Imbriquer et garder ensemble les données que vous allez toujours récupérer ensemble.*

En MySQL, vous séparez les données quoi qu'il arrive, même si en pratique vous allez toujours résoudre certaines jointures.

En MongoDB, on efface les jointures “inutiles” et on *embarque* à la place.

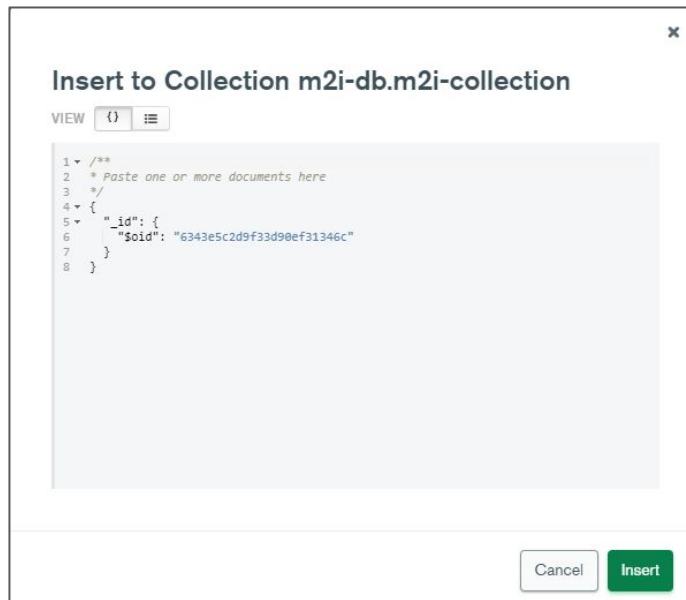
# Manipuler MongoDB

Via MongoDB Compass, dans votre collection :  
*ADD DATA -> Insert Document*



# Manipuler MongoDB

Vous pouvez insérer le JSON de votre choix :



The screenshot shows a dialog box titled "Insert to Collection m2i-db.m2i-collection". It has a "VIEW" button with a toggle switch set to "JSON" and a list icon. The main area is a text editor with line numbers 1 through 8. The text inside is a JSON document: 

```
1 /**
2  * Paste one or more documents here
3  */
4 {
5   "_id": {
6     "$oid": "6343e5c2d9f33d90ef31346c"
7   }
8 }
```

 At the bottom right, there are two buttons: "Cancel" and "Insert".



# Manipuler MongoDB

Remarquez que vous pouvez insérer plusieurs JSON avec des champs différents dans la même collection.

MongoDB n'impose pas de structure sur vos collections.

En pratique, nous allons nous en imposer une nous-même et stocker des documents similaires dans une même collection.

# Manipuler MongoDB

Petit rappel :



# Manipuler MongoDB

Insérer des données dans MongoDB via Compass à l'aide d'un fichier :

- Une ligne par document
- `"_id": { "$oid": "votre identifiant" }}` pour un identifiant explicite

# Manipuler MongoDB

Exercices : insérer des données

# MongoSH

Connectez vous via MongoSH :

```
Please enter a MongoDB connection string (Default: mongodb://localhost/): _
```

```
mongodb://mongo:mongo@localhost:27017?authSource=admin
```

# MongoSH

Une fois connecté :

```
-----  
Enable MongoDB's free cloud-based monitoring service, which will then receive and display  
metrics about your deployment (disk utilization, CPU, operation statistics, etc).  
  
The monitoring data will be available on a MongoDB website with a unique URL accessible to you  
and anyone you share the URL with. MongoDB may use this information to make product  
improvements and to suggest MongoDB products and deployment options to you.  
  
To enable free monitoring, run the following command: db.enableFreeMonitoring()  
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()  
-----  
test>
```

# MongoSH

Pour sélectionner une base de données :

```
use <database-name>
```

Exemple :

```
use m2i-db
```

# MongoSH

Pour afficher le contenu d'une collection :

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

Exemple :

```
db.simple_entity.find()
```



# MongoSH

Exécuter un fichier JavaScript :

- Créez un fichier JavaScript “test.js”
- Ajoutez le contenu suivant dans le fichier :

```
const result = db.simple_entity.find()  
printjson(result)
```

- Exécutez le fichier depuis MongoSH (attention aux “\” ) :

```
m2i-db> load("C:\\Users\\Xahellz\\Formation\\test.js")
```

# MongoSH

A partir de maintenant, nous utiliserons des fichiers.

```
m2i-db> load("C:\\Users\\Xahellz\\Formation\\test.js")
[
  {
    _id: ObjectId("634676219235d0359d09a879"),
    name: 'Simple 1',
    price: 39.99
  },
  {
    _id: ObjectId("634676219235d0359d09a87a"),
    name: 'Simple 2',
    price: 29.99
  },
  {
    _id: ObjectId("634676219235d0359d09a87b"),
    name: 'Simple 3',
    price: 19.99,
    disabled: true
  },
  {
    _id: ObjectId("634676219235d0359d09a87c"),
    name: 'Simple 3',
    priceWithTypo: 4.99,
    disabled: false
  }
]
true
```

# Queries

Pour exécuter une recherche sur une collection avec MongoDB :

```
db.collection.find({ /* query */ })
```

La query prend en entrée un objet JavaScript.

# Queries, test d'égalité

```
const result = db.simple_entity.find(  
{  
  "name" : "Simple 1"  
})  
  
printjson(result)
```

## “and”, on ajoute des champs

```
const result = db.simple_entity.find(  
  {  
    "name" : "Simple 1",  
    "price" : 39.99  
  })  
  
printjson(result)
```

# “\$or”, un tableau de sous-requêtes

```
const result = db.simple_entity.find(  
{  
  "$or" : [  
    { "name" : "Simple 1" },  
    { "price" : 29.99 }  
  ]  
})  
printjson(result)
```

# “\$or”, un tableau de sous-requêtes

```
"$or" : [  
    <subquery 1>,  
    <subquery 2>,  
    ...  
    <subquery n>  
]
```

# Mélanger les “and” et les “or”

## Expression booléenne :

`a == 1 && (b == 2 || c == 3)`

## Équivalent MongoDB :

```
{  
  "a" : 1,  
  "$or" : [  
    "b" : 2,  
    "c" : 3  
  ]  
}
```



## “\$gt”, l’opérateur >

```
const result = db.simple_entity.find(  
{  
  "price" : { "$gt" : 20 }  
})  
  
printjson(result)
```

## “\$lt”, l’opérateur <

```
const result = db.simple_entity.find(  
  {  
    "price" : { "$lt" : 20 }  
  })  
  
printjson(result)
```

# \$lt et \$gt pour un between

```
const result = db.simple_entity.find(  
  {  
    "price" : {  
      "$gt" : 10,  
      "$lt" : 30  
    }  
  })  
printjson(result)
```

# \$lte et \$gte pour l'égalité incluse

```
const result = db.simple_entity.find(  
  {  
    "price" : {  
      "$gte" : 19.99  
    }  
  })  
  
printjson(result)
```

Greater **T**han or **E**quals

# Egalité de tableau

```
const result = db.array_entity.find(  
  {  
    "values" : [3, 0, 0, 0]  
  })  
  
printjson(result)
```

# Tester la présence dans le tableau

```
const result = db.array_entity.find(  
{  
  "values" : { "$all" : [2, 1] }  
})  
  
printjson(result)
```

Il faut les valeurs 2 et 1 dans le tableau. Le tableau peut avoir plus d'éléments.

# Tester la présence dans le tableau

```
const result = db.array_entity.find(  
{  
  "values" : { "$in" : [2, 1] }  
})  
  
printjson(result)
```

Il faut les valeurs 2 ou 1 dans le tableau. Le tableau peut avoir plus d'éléments.

# Plusieurs tests, on ajoute des champs

```
const result = db.array_entity.find(  
{  
  "values" : { "$gte" : 3, "$lte" : 3 }  
})  
  
printjson(result)
```

Un élément supérieur ou égal à 3, un élément inférieur ou égal à 3.



# Plusieurs tests sur le même élément

```
const result = db.array_entity.find(  
{  
  "values" : { "$elemMatch" : { "$gte" : 3, "$lte" : 3 } }  
})  
  
printjson(result)
```

Un élément supérieur ou égal à 3 **ET** inférieur ou égal à 3.  
(donc ici, égal à 3)

# Tester si un champ existe

```
const result = db.simple_entity.find(  
{  
  "disabled" : { "$exists" : true }  
})  
  
printjson(result)
```

Ne renvoie que les documents qui ont un champ “disabled” (peu importe la valeur).

# Tester un embedded

```
const result = db.embedded_entity.find(  
{  
  "embedded.value" : 0.2  
})  
  
printjson(result)
```

Teste si le champ value de l'attribut "embedded" vaut 0.2.

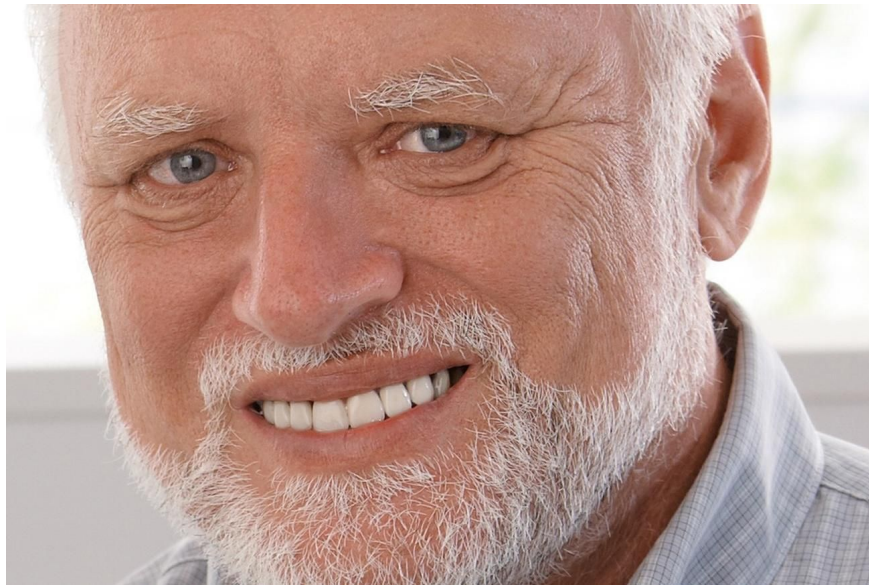
# Tester un référence

On veut récupérer les lignes où leur entité référencée a une certaine valeur.

SQL : on ferait une jointure.

En MongoDB :

# Tester un référence



# Tester un référence

On ne fait pas.

On utilise du code plus compliqué où on résout les références nous-même, “à la main”.

Mais pas de jointure comme en SQL, pas de clause EXISTS, etc.

MongoDB n'est pas prévu pour cela.

# Query - Exercices

Exercices.

# Valider le schéma

En SQL, vous avez l'habitude de vérifier vos données :

- Colonnes nullable / pas nullable
- Clés étrangères vers une ligne qui existe
- Min/max
- Énumérations
- etc.



# Valider le schéma

En MongoDB, nous pouvons faire la même chose :

**m2i-db.embedded\_entity**

41DOCUMENTSINDEXES

DocumentsAggregationsSchemaExplain PlanIndexesValidation

Validation Action ⓘERRORValidation Level ⓘSTRICT

```
1 {
2   $jsonSchema: {
3     /* rules */
4   }
5 }
```

Validation modified

CANCELUPDATE

# Valider le schéma - Required

Forcer la présence de champs :

```
{  
  $jsonSchema: {  
    required: ['name', 'price']  
  }  
}
```

# Valider le schéma - Properties

Properties va contenir les tests sur les champs

```
{  
  $jsonSchema: {  
    properties: {  
      <field> : { /** tests */ }  
    }  
  }  
}
```

<field> est le nom du champ à tester

# Valider le schéma - Typage

Forcer le type d'un champ :

```
{
  $jsonSchema: {
    properties: {
      name: {
        bsonType: 'string'
      }
    }
  }
}
```

Les types :

<https://www.mongodb.com/docs/manual/reference/bson-types/>

# Valider le schéma - Min/max

Forcer le type d'un champ :

```
{  
  $jsonSchema: {  
    properties: {  
      price: {  
        minimum: 0,  
        maximum: 9999  
      }  
    }  
  }  
}
```

(ne fait rien sur les string)

# Valider le schéma - Queries

A la place du jsonSchema, vous pouvez mettre une query pour valider votre schéma.

Est valide n'importe quel document qui serait retourné par la query :

```
{
  name: {
    $in: [
      'Complex 1',
      'Complex 2'
    ]
  }
}
```

# MongoDB - Indexes

Les indexes servent à réduire le temps d'exécution de vos requêtes.

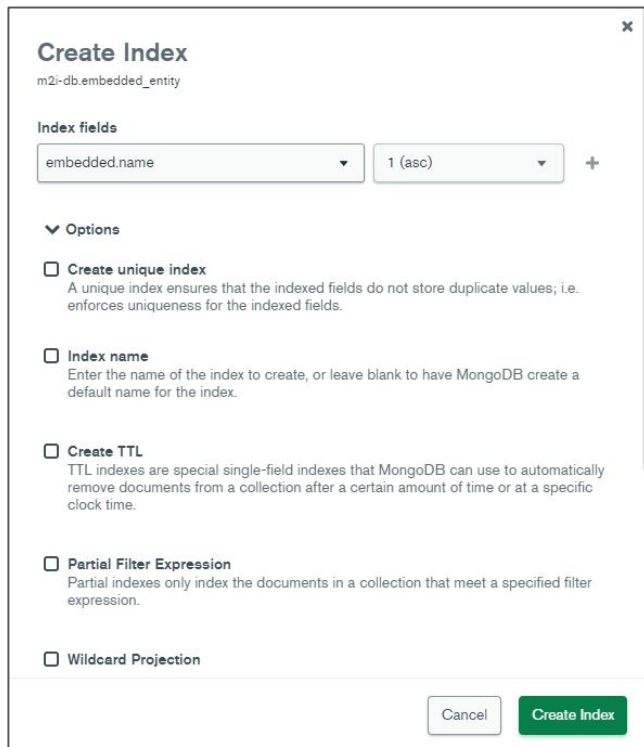
Un index a deux désavantage :

- Il utilise de la mémoire
- Il rend l'insertion de nouveaux documents plus lent

On indexe donc ce qui est utile à indexer. C'est-à dire les champs sur lesquels nous allons effectuer des tests lors des queries.

# MongoDB - Indexes

Pour créer un index, on peut passer par Compass via l'onglet Index :



The screenshot shows the 'Create Index' dialog box in MongoDB Compass. The dialog is titled 'Create Index' and shows the database 'm2i-db' and collection 'embedded\_entity'. Under 'Index fields', 'embedded.name' is selected with an ascending order of '1 (asc)'. The 'Options' section is expanded, showing several unchecked checkboxes: 'Create unique index' (with a description), 'Index name' (with a description), 'Create TTL' (with a description), 'Partial Filter Expression' (with a description), and 'Wildcard Projection'. At the bottom right, there are 'Cancel' and 'Create Index' buttons.

**Create Index**  
m2i-db.embedded\_entity

**Index fields**

embedded.name 1 (asc) +

**Options**

- ☐ **Create unique index**  
A unique index ensures that the indexed fields do not store duplicate values; i.e. enforces uniqueness for the indexed fields.
- ☐ **Index name**  
Enter the name of the index to create, or leave blank to have MongoDB create a default name for the index.
- ☐ **Create TTL**  
TTL indexes are special single-field indexes that MongoDB can use to automatically remove documents from a collection after a certain amount of time or at a specific clock time.
- ☐ **Partial Filter Expression**  
Partial indexes only index the documents in a collection that meet a specified filter expression.
- ☐ **Wildcard Projection**

Cancel Create Index



# MongoDB - Indexes

Index unique : Chaque valeur est présente une seule fois

Index name : pour choisir nous-même le nom de l'index

TTL : donne une expiration à chaque document (exemple : un document vie 60 min)

Partial index : un index qui n'est appliqué que sur les lignes qui satisfont une query spécifique (le but principal est de ne pas tout indexer pour économiser la mémoire)

Wildcard index : `{ "fieldA.$*" : 1 }`, index tous les champs d'un embedded

# MongoDB - Indexes

Sparse index: n'indexe pas les documents qui n'ont pas le champ indexé

# MongoDB - Aggregation

L'agrégation permet de calculer des résultats sur des ensembles (somme/moyenne/etc.) et permet de transformer de la donnée.

L'agrégation est similaire au concept fonctionnel “map/filter/reduce” que l'on peut trouver dans tous les langages.

# MongoDB - Aggregation

```
const result = db.simple_entity.aggregate([
  { <operationName1> : { /* params */ } },
  { <operationName2> : { /* params */ } },
  { <operationName3> : { /* params */ } }
])

printjson(result)
```

# \$group

Calculer une somme se fait avec une opération \$group :

```
const result = db.simple_entity.aggregate([
  {
    $group : { _id = '$name', totalPrice : { $sum : '$price' } }
  }
])

printjson(result)
```

# \$group

```
$group : { _id = '$name', totalPrice : { $sum : '$price' } }
```

\$group définit une opération de groupement (somme/moyenne/count/etc.)

Deux paramètres :

- L'identifiant pour grouper les éléments `_id`
- Le nom de la colonne à créer et contenant le résultat

# \$group

```
$group : { _id = '$name', totalPrice : { $sum : '$price' } }
```

'\$<champ>' permet de préciser que l'on mentionne un champ, et non pas une valeur.

On groupe via le champ "name" et l'on somme sur la colonne "price".

# \$group

```
$group : { _id = '$name', totalPrice : { $sum : '$price' } }
```

Lorsque l'on crée une nouvelle colonne, plusieurs opérations possibles :

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

\$sum, \$avg, \$max, \$first, \$accumulator...



# \$match

\$match permet de filtrer, il prend en paramètre une query. Exemple, filtrer par prix :

```
const result = db.simple_entity.aggregate([
  {
    $match : { 'price' : { $gte: 5 } }
  }
])

printjson(result)
```

# MongoDB - Aggregation

Avec les aggregations, nous pouvons faire une succession d'opérations :

```
const result = db.simple_entity.aggregate([
  { $match : { /* params */ } },
  { $group : { /* params */ } },
  { $match : { /* params */ } },
  { $group : { /* params */ } }
])
printjson(result)
```

# MongoDB - Aggregation

Par exemple :

Sur articles dont le prix est supérieur à 10€, calculer la somme des prix par groupe d'article.

Puis calculer la moyenne sur cette somme par groupe d'article.

Il s'agit d'une "pipeline".

# MongoDB - Aggregation

L'agrégation permet également de faire des “update” sur nos données, via une opération nommée “\$merge”.

Mais il s'agit d'une opération particulièrement complexe, nous la verrons plus tard si l'on a le temps.

# MongoDB et Spring

Pour intégrer MongoDB au sein de Spring :

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-data-mongodb</artifactId>  
</dependency>
```

# MongoDB et Spring

Attention, retirez le scope “test” à jackson (il est utilisé par MongoDB en runtime) !

```
<dependency>  
  <groupId>com.fasterxml.jackson.core</groupId>  
  <artifactId>jackson-databind</artifactId>  
</dependency>
```

# MongoDB et Spring

application.properties :

```
spring.data.mongodb.uri=mongodb://mongo:mongo@localhost:27017  
spring.data.mongodb.database=m2i-db
```

# MongoDB et Spring

Autoriser les Repository de MongoDB :

```
@Configuration  
@EnableMongoRepositories  
public class MongoConfig {  
  
}
```



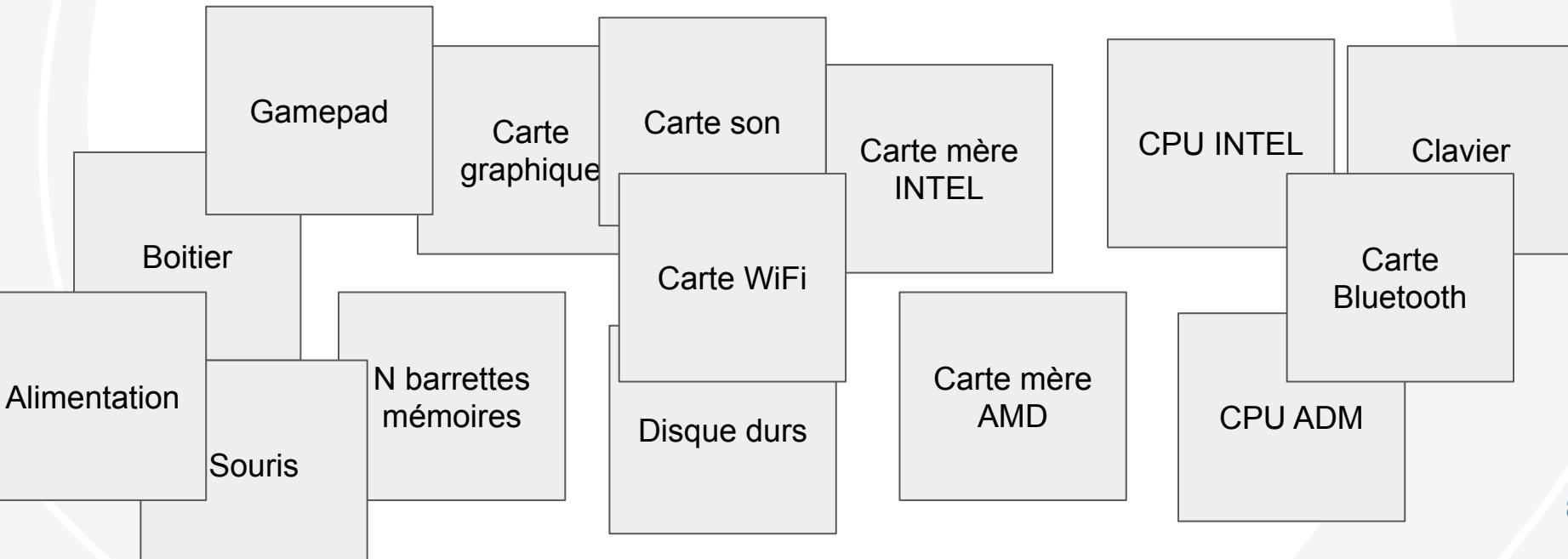
# Design pattern Builder

Bonus : focus le design pattern Builder

# Design pattern Builder

## Le problème :

Certains objets sont complexes à construire. Exemple, créer un ordinateur.



# Design pattern Builder

## Le problème :

Certains objets sont complexes à construire. Exemple, créer un ordinateur.

```
new Computer(computerCase, cpu, motherboard, wifiCard,  
bluetoothCard, keyboard, mouse, gamepads, gpu, mems,  
soundCard...)
```

Chaque paramètre peut être nul, le constructeur est illisible.

# Design pattern Builder

## La solution :

Créer une autre classe “Builder” qui va masquer le constructeur à l'utilisateur et proposer une interface plus flexible.

```
class Builder
{
    public void addComputerCase(ComputerCase ...) { ... }
    public void addMotherboard(Motherboard ...) { ... }

    public Computer build() { return new Computer(...); }
}
```

# Design pattern Builder

## Utilisation :

```
Builder builder = new Builder();
```

```
builder.addComputerCase(computerCase);  
builder.addMotherboard(motherBoard);
```

```
Computer computer = builder.build();
```

# Design pattern Builder

## En syntaxe fonctionnelle :

```
class Builder
{
    public Builder addComputerCase(ComputerCase ...) { ... return this; }
    public Builder addMotherboard(Motherboard ...) { ... return this; }

    public void build() { return new Computer(...); }
}
```

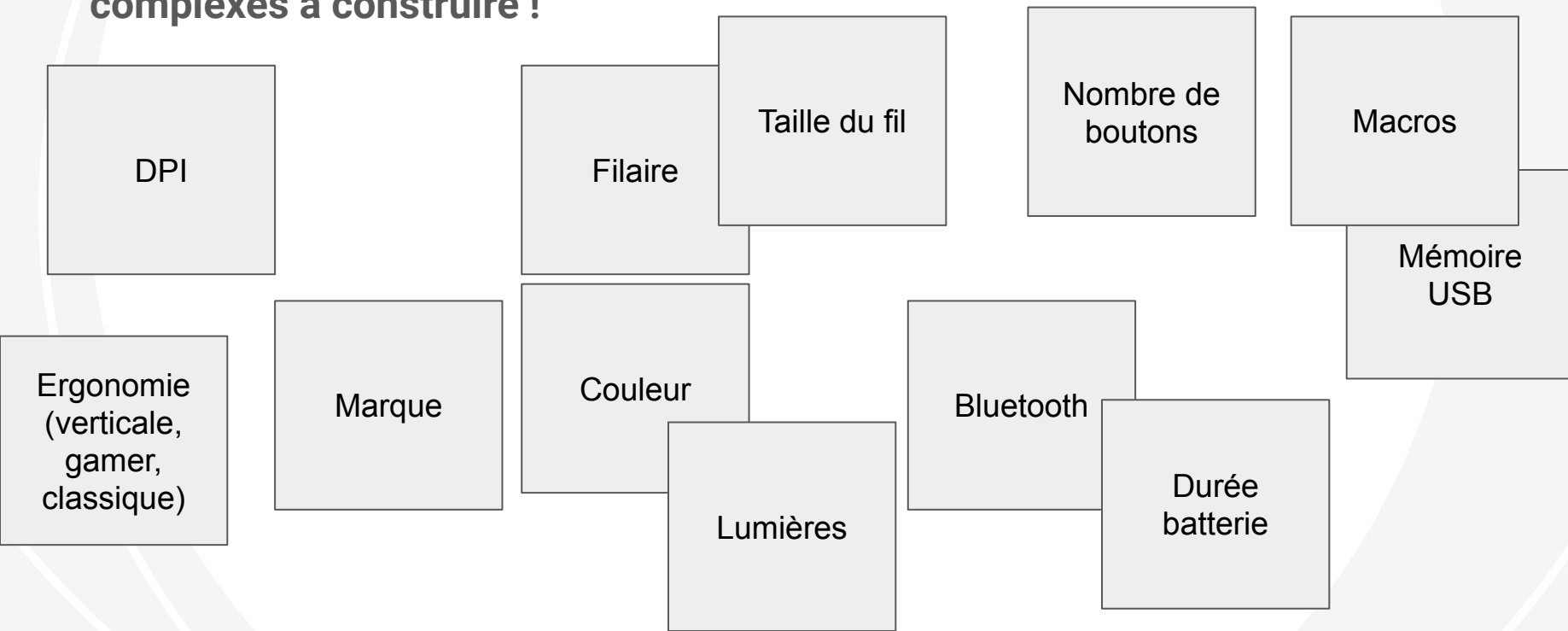
# Design pattern Builder

**En syntaxe fonctionnelle :**

```
Builder builder = new Builder()  
    .addComputerCase(computerCase)  
    .addMotherboard(motherBoard)  
    .build();
```

# Design pattern Builder

Deuxième problème, les autres éléments sont peut-être également complexes à construire !





# Design pattern Builder

## La solution :

Nous allons imbriquer des builders.

```
public class ComputerBuilder
{
    private CPUBuilder cpuBuilder = new CPUBuilder();
    public CPUBuilder buildCPU() { ... return new CPUBuilder(this); }
    public MouseBuilder buildMouse() { ... return new MouseBuilder(this); }

    public Computer build() { return new Computer(cpuBuilder.build(), ...); }
}
```

# Design pattern Builder

## La solution :

Nous allons imbriquer des builders

```
public class MouseBuilder
{
    private ComputerBuilder ...;

    public MouseBuilder setColor(Color color) { ... return this; }

    public ComputerBuilder and() {
        return computerBuilder;
    }
}
```

# Design pattern Builder

## Exemple d'utilisation :

```
ComputerBuilder builder = new ComputerBuilder()  
    .buildMouse()  
        .setColor(Color.RED)  
        .and()  
    .buildCPU()  
        .setBrand(Brands.INTEL)  
        .and()  
    .build()
```

# Design pattern Builder

## Exercices