U3 MongoDB i Java

Accés a dades DAM

Continguts

1. Recursos necessaris
2. Connexió al servidor
3. Bases de dades i col·leccions
4. Classe org.bson.Document
5. Insercions
6. Consultes
7. FindIterable
8. Filtres
9. Sort
10. Projections
11. Aggregate
12. Modificacions
13. Eliminacions

1. Recursos necessaris

El driver de Mongodb per Java es pot descarregar de la url

https://mongodb.github.io/mongo-java-driver/ amb dues versions:

- Mongodb Driver: Operacions síncrones. El que utilitzarem noltros:

  mongodb-driver-sync-3.10.0.jar

- Mongodb Async Driver: Operacions asíncrones.

  mongodb-driver-async-3.10.0.jar

Tots dos necessiten dos jars addicionals:

- **mongodb-driver-core**: classes bàsiques per el funcionament del driver.

  Necessari per elaborar api's pròpies per atacar mongodb.

- **bson**: classes per tractar amb documents bson.

https://www.mongodb.com/docs/drivers/java/sync/current/quick-start/#quick-start

2. Connexió al servidor

EL servidor necessita estar configurat per acceptar connexions des de l'exterior.Per ferho s'ha de modificar el fitxer mogod.conf i posar la propietat bindIP=0.0.0.0

L'objecte MongoClient representa la connexió amb el servidor. En realitat és un pool de connexions. Només es necessita una instància encara que es treballi amb diversos Threads. El constructor té diverses sobrecàrregues:

- Per connectar l'aplicació a localhost al port (per defecte, 27017)

```
MongoClient mongoClient = MongoClients.create()
```

- Per connectar l'aplicació al host especificat al port per defecte:

```
MongoClient mongoClient = MongoClients.create(
        MongoClientSettings.builder()
        .applyToClusterSettings(builder ->
                builder.hosts(Arrays.asList(new ServerAddress("nom o ip del host"))))
        .build());
```

3. Bases de dades i col·leccions

Per connectar l'aplicació al host i al port especificats:

```
MongoClient mongoClient = MongoClients.create(
        MongoClientSettings.builder()
                .applyToClusterSettings(builder ->
                        builder.hosts(Arrays.asList(new ServerAddress("host", 27018))))
                .build());
```

També podem utilitzar una url, pot ser la més còmoda:

```
MongoClient mongoClient = MongoClients.create("mongodb://hostOne:27017");
```

Bases de dades i col·leccions

Una vegada tenim la instància de *MongoClient* la utilitzarem per obtenir accés a la base de dades:

```
MongoDatabase db = mongoClient.getDatabase("SPADD");
```

A partir de la base de dades tendrem accés a les col·leccions que conté.

```
MongoCollection<Document> col = db.getCollection("AspirantsComplet");
```

La col·lecció es representa com una instància de la classe *MongoCollection* que conté objectes *org.bson.Document*.

Si la col·lecció no existeix, en insertar-hi documents la crearà. Podem crear-la explícitament, amb opcions o sense:

```
db.createCollection("newCollection");
```

```
ValidationOptions collOptions = new ValidationOptions().validator(
        Filters.or(Filters.exists("email"), Filters.exists("phone")));
database.createCollection("contacts",
        new CreateCollectionOptions().validationOptions(collOptions))
```

Podem esborrar una col·lecció:

```
col.drop();
```

4. Classe org.bson.Document

La classe Document representa un dels documents de la col·lecció. Necessitarem crear documents per especificar filtres, projeccions,...

Té tres constructors:

- **Document()**
- **Document(String, Object)**: afegeix una propietat al document amb el nom de l'string i el valor del document.
- **Document(Map<String, Object>)**: Inicialitza el document amb les propietats definides al mapa.

Alguns dels mètodes dels que disposa:

- **.append(String, Object):** Afegeix la propietat al document. Torna el document.
- **.get(nomPropietat)**: Torna l'objecte associat a la propietat. També tenim getBoolean, getInteger, getString, ... que tornen el valor en aquest format si es possible. getList(nomPropietat, ClasseElements.class)
- **.toJson()**: torna la cadena amb el json del document.

4. Classe org.bson.Document

Per representar un array necessitam qualsevol implementació de List, per exemple:

```
ArrayList dades=new ArrayList();
dades.add("v3.2");
dades.add(3.0);
dades.add("v2.6");
```

O bé

```
Arrays.asList("v3.2", 3.0, "v2.6")
```

Per representar un objecte hem d'incloure un document:

```
.append("info", new Document("x", 203).append("y", 102));
```

4. Classe org.bson.Document

Per exemple, el següent document

```
{
    "name" : "MongoDB",
    "type" : "database",
    "count" : 1,
    "versions": [ "v3.2", "v3.0", "v2.6" ],
    "info" : { x : 203, y : 102 }
}
```

Es representaria com:

```
Document doc = new Document("name", "MongoDB")
        .append("type", "database")
        .append("count", 1)
        .append("versions", Arrays.asList("v3.2", "v3.0", "v2.6"))
        .append("info", new Document("x", 203).append("y", 102));
```

5. Insercions

La classe **MongoCollection** disposa dels mètodes **insertOne** i **insertMany** per fer insercions a la base de dades.

- **.insertOne** espera com a paràmetre un objecte Document. S'ha de crear el document, afegir-li totes les seves propietats i subdocuments i passar-lo al mètode.
- **.insertMany** espera com a paràmetre una llista d'objectes Document.

Exemples

```
collection.insertOne(new Document().append("codi", "123").append("nom", "Joan"));
```

Utilitzant un Map

```
HashMap<String, Object> propietats=new HashMap<>();
propietats.put("codi", "123456");
propietats.put("nom", "Josep");
mongo.collection.insertOne(new Document(propietats));
```

Una inserció múltiple:

```
List<Document> documents = new ArrayList<Document>();for (int i = 0; i < 100; i++) {
        documents.add(new Document("i", i));
}
collection.insertMany(documents);
```

6. Consultes

La classe **MongoCollection** disposa del mètode find que torna un objecte de la classe **FindIterable** que es pot recòrrer amb un for. No és massa recomanable perquè en realitat és un cursor que hauríem de tancar.

```
FindIterable<Document> find = collection.find();
for (Document doc : find) {
        System.out.println(doc);
}
```

Una manera millor de fer-ho és amb el MongoCursor:

```
MongoCursor<Document> cursor = collection.find().iterator();
try {
        while (cursor.hasNext()) {
                System.out.println(cursor.next().toJson());
        }
} finally {
        cursor.close();
}
```

7. FindIterable

FindIterable ens proporciona una sèrie de mètodes que s'ocupen adequadament del cursor que implementa:

- **.forEach(Block)**: Aplica el block a cada un dels resultats del cursor. Deprecated,

  Utilitza el MongoCursor o el foreach de Iterable:

```
collection.find().forEach((Consumer<? super Document>) document-
>System.out.println(document.toJson()));
```

- **.into(col·leccioJava)**: Rep una col·lecció, per exemple un ArrayList, i el torna ple amb les dades del cursor.

```
ArrayList<Document> llista = new ArrayList<>();
collection.find().into(llista);
```

- **.first()**: Torna el primer element del cursor.

```
Document first=collection.find.first();
```

7. FindIterable

- **.find(Bson)**: El paràmetre és un document amb les condicions del filtre. Torna un FindIterable.

```
proves.find(new Document("name", "PostgreSQL")).forEach((Consumer<? super
Document>) System.out::println);
```

```
proves.find(new Document("type", "database").append("count",new
Document("$gte",3)))
```

  Mes endavant veurem una altra manera de fer-ho utilitzant Filters.

- **.limit(int n)**: Limita el nombre de resultats. Torna un *FindIterable*.
- **.skip(int n)**: Ignora els n primers resultats. Torna un *FindIterable*.
- **.projection(Bson)**: crea una projecció amb els detalls especificats al document que rep com a paràmetre . Torna un FindIterable.

```
proves.find(new Document("type", "database"))
.projection( new Document("_id",0).append("name",1).append("type",1))
.forEach((Consumer<? super Document>) System.out::println);
```

7. FindIterable

La cosa es simplifica amb els mètodes de la classe **Projections**.

- **.sort(Bson):** Ordena els resultats segons el criteri especificat al document bson passat com a paràmetre. Torna un FindIterable.

```
proves.find(new Document("type", "database"))
.projection( new Document("_id",0).append("name",1).append("type","database"))
.sort(new Document("name",1)
.forEach((Consumer<? super Document>) System.out::println);
```

En lloc del document es poden utilitzar els mètodes de la classe Sorts.

La majoria d'aquests mètodes tornen un altre FindIterable, de manera que es poden encadenar fent un *pipeline*.

8. Filtres

Dins find podem especificar el paràmetre que filtra les dades. Tenim dues opcions:

- Crear un document amb els criteris de cerca: El paràmetre és un document amb les condicions del filtre.

```
collection.find(
 new Document("stars", new Document("$gte", 2)
 .append("$lt", 5))
 .append("categories", "Bakery")).forEach(printBlock);
```

- Utilitzar la classe Filters. Normalment s'utilitza amb un import static.

```
collection.find(and(gte("stars", 2), lt("stars", 5), eq("categories", "Bakery"))).
forEach(printBlock);
```

Els dos exemples són l'equivalent de la consulta que podríem fer al shell:

```
find({stars:{$gte:2,$lt:5},"categories":"Bakery"})
```

Alguns dels filtres que es poden aplicar són: all, and, eq, exists, gt, gte, in, lt, lte, ne, nin, nor, not, or, regex, ...

9. Sort

Després del find podem especificar el criteri d'ordenació amb sort. Podem utilitzar la classe **Sorts**. Normalment s'utilitza amb un import static.

```
collection.find(and(gte("stars", 2), lt("stars", 5), eq("categories", "Bakery"))).
sort(ascending("stars")).
forEach(printBlock);
```

Ordenaria els documents per ordre ascendent segons el camp stars. Els mètodes de Sorts accepten o bé una llista amb el nom dels camps, o bé varargs.

- **.ascending(Llista o varargs)**: Ordena de forma ascendent segons els camps especificats.
- **.descending(Llista ovarargs)**: Ordena de forma descendent segons els camps especificats.
- **.orderBy(varargs de sorts)**: Combina múltiples sorts.

~~Podem posar diversos sort amb el seu ascending o descending, però això farà que el segon reordeni el resultat del primer.~~ **Si el que volem es combinar-los**, per exemple ordenar de forma ascendent per els llinatges i descendent per el nom **hem d'utilitzar orderBy**.

10. Projections

Per defecte les consultes tornen tots els camps dels documents. Podem fer una projecció per especificar aquesta llista de camps.

- Crear un document amb els camps a 1 si els volem incloure i a 0 si no:

```
collection.find().projection(
 new Document("stars", 1).
.append("_id", 0)).forEach(printBlock);
```

- Utilitzar la classe **Projections**. Normalment s'utilitza amb un import static.

```
collection.find().projection(fields(include("stars","category"), excludeId()))
forEach(printBlock);
```

Algunes de les projeccions que es poden utilitzar són:

- .include(Llista o vararg): Passam per paràmetre els noms dels camps a incloure
- .exclude(Llista o vararg):Passam per paràmetre els noms dels camps a excloure
- .excludeId(): Sense paràmetres, per excloure l'id.
- .fields(varargs): combina els include, exclude, ... que li passam com a arguments.

11. Aggregate

La classe **MongoCollection** té el mètode aggregate que permet executar una llista d'etapes, o *aggregation stages*, de manera que cada una rebi la sortida de l'anterior.

Podem crear aquesta llista amb Arrays.asList()

```
collection.aggregate(Arrays.asList(skip(10),limit(10))).forEach(printComplet);
```

També podem crear una llista d'objectes Bson i passar-la com a argument

```
ArrayList<Bson> llista=new ArrayList<>();
llista.add(skip(10));
llista.add(limit(10));
collection.aggregate(llista).forEach(printComplet);
```

La classe **Aggregates** defineix els mètodes static que es poden utilitzar com a etapes. Entre d'altres defineix count, group, limit, match, out, project, skip, sort, unwind, ...

12. Modificacions

La classe **MongoCollection** disposa dels mètodes *updateOne* i *updateMany* per fer modificacions a la base de dades. Tots dos tenen dos paràmetres, el filtre i les modificacions.

- .updateOne Només actualitza el primer document que torni el filtre encara que en torni més.
- .updateMany Actualitza tots els documents que torna el filtre.

Tots dos tornen un objecte UpdateResult que conté el resum de l'update. Per exemple .**getMatchedCount**() torna quants documents han coincidit amb el filtre i .**getModifiedCount**() els que ha modificat.

Per especificar el filtre tenim els mètodes de la classe **Filters**, i per les modificacions els de la classe **Updates**.

```
collection.updateMany(eq("stars", 2),
combine( set("stars", 0), currentDate("lastModified")));
```

Si volem utilitzar més d'un mètode hem d'utilitzar combine. Altres mètodes de la classe update són currentDate, inc, max, min, push, pushEach, rename, set, unset, ...

13. Eliminacions

La classe **MongoCollection** disposa dels mètodes *deleteOne* i *deleteMany* per fer eliminacions a la base de dades. Tots dos tenen un sol paràmetre, el filtre que selecciona els documents a eliminar.

- .**deleteOne** Només elimina el primer document que torni el filtre encara que en torni més.
- .**deleteMany** Elimina tots els documents que torna el filtre.

Tots dos tornen un objecte DeleteResult que conté el resum del delete. Per

exemple .**getDeletedCount**() torna els documents que ha esborrat.

Per especificar el filtre tenim els mètodes de la classe **Filters**.

```
collection.deleteOne(eq("_id", new ObjectId("57506d62f57802807471dd41")));
collection.deleteMany(eq("stars", 4));
```

```
WARNING: SLF4J not found on the classpath.  Logging is disabled for the
'org.mongodb.driver' component
```

```
            <dependency>
                <groupId>org.apache.logging.log4j</groupId>
                <artifactId>log4j-slf4j-impl</artifactId>
                <version>2.17.1</version>
            </dependency>
```