MongoDB

Excerpts from

"*The Little MongoDB Book*"

Karl Seguin

- Download MongoDB from https://www.mongodb.com/download-center/community
- Install (double-click) and follow the steps. These videos may help:

https://www.youtube.com/watch?v=kPKwJWr_9TM

https://www.youtube.com/watch?v=gkCnXcxHC4o

- Install it preferably without selecting the option "*Install MongoD as a Service*"
- Create data\db at root directory

- Navigate to the MongoDB bin subfolder (in MongoDB 5.0 it is at C:\Program Files\MongoDB\Server\5.0\bin), mongod is the server process and mongo is the client shell.
- Double-click on mongod (the server process, if it was not chosen as a service)

After launching it, do not close it!

- Double-click on mongo (without the *d*) which will connect a shell to your running server.
- You can also use a more friendly ☺ manager, for example, MongoDB Compass or NoSQL Manager among others:

https://www.mongodbmanager.com

- MongoDB has the same concept of a database with which you are likely already familiar (a database schema).
- Within a MongoDB instance you can have zero or more databases.
- A database can have zero or more collections. A collection shares enough in common with a traditional table
- Collections are made up of zero or more documents. A document can safely be thought of as a row.
- A document is made up of one or more fields, which you can probably guess are a lot like columns.

- Indexes in MongoDB function mostly like their RDBMS counterparts.

To recap, MongoDB is made up of databases which contain collections. A collection is made up of documents. Each document is made up of fields. Collections can be indexed, which improves lookup and sorting performance.

- Why use new terminology (collection vs. table, document vs. row and field vs. column)?
- While these concepts are similar to their relational database counterparts, they are not identical.

  The core difference comes from the fact that relational databases define columns at the table level whereas a document-oriented database defines its fields at the document level

- That is: each document within a collection can have its own unique set of fields.

- Ultimately, the point is that a collection is not strict about what goes in it (it is schema-less).
- Let us get hands-on. Go ahead and enter db.help(), you will get a list of commands that you can execute against the db object.
- First we will use the global use helper to switch databases, so go ahead and enter **use learn**. It does not matter that the database does not really exist yet.

db is a variable which represents current database

- The first collection that we create will also create the actual learn database.
- Now that you are inside a database, you can start issuing database commands, like db.getCollectionNames()
- You could get an empty array [ ].
- Try these commands:
  - db
  - show dbs

- We can simply insert a document into a new collection. To do so, use the insert command, supplying it with the document to insert:

```
db.unicorns.insert({name: 'Aurora', gender: 'f', weight: 450})
```

Now again: `db.getCollectionNames()`

- You can now use the find command against unicorns to return a list of documents:

db.unicorns.find()

Try also: db.unicorns.find().pretty()

- Notice that, in addition to the data you specified, there is an _id field. Every document must have a unique _id field.
- You can either generate one yourself or let MongoDB generate a value for you which has the type ObjectId. Most of the time you will probably want to let MongoDB generate it for you.

- Now insert:

```
db.unicorns.insert({name: 'Leto',
gender: 'm',
home: 'Arrakeen',
worm: false})

db.unicorns.find()
```

- There is one practical aspect of MongoDB you need to have a good grasp of before moving to more advanced topics: query selectors.
- First, remove what we have put so far in the unicorns collection via:

db.unicorns.**remove**({})

Try also this: db.unicorns.**drop**()

Nota: remove y drop son análogos a delete y drop de SQL…

- Issue the inserts in the file **unicorns.txt (ver menú Varios)**

db.unicorns.find()

- **Nota**: Un archivo también se puede cargar desde un archivo mediante mongoimport, pero a partir de MongoDB 4.4 hay que instalar también MongoDB Database Tools:

https://www.mongodb.com/docs/database-tools/installation/installation

Antes de la 4.4 se ejecutaba desde el directorio bin:

```
mongoimport --jsonArray --db learn --collection unicorns --file="C:/Tempi/otrosunicornios.txt"
```

Donde el archivo otrosunicornios.txt, ubicado en la carpeta Tempi en este ejemplo, está formado como se muestra en la siguiente diapositiva. Note que se usan comillas dobles para los nombres de los campos y para sus valores (strings) y que los documentos están en un arreglo, el cual inicia por [y termina con].

```
[
{
"name": "Lisa",
"gender": "f"
},
{
"name": "Dino",
"gender": "m",
"vampires": 33
}
]
```

- Now that we have data, we can master selectors.
- {field: value} is used to find any documents where field is equal to value.
- {field1: value1, field2: value2} is how we do an AND statement. The special $lt, $lte, $gt, $gte and $ne are used for less than, less than or equal, greater than, greater than or equal and not equal operations.

- For example, to get all male unicorns that weigh more than 700 pounds, we could do:

```
db.unicorns.find({gender: 'm', weight: {$gt: 700}})
```

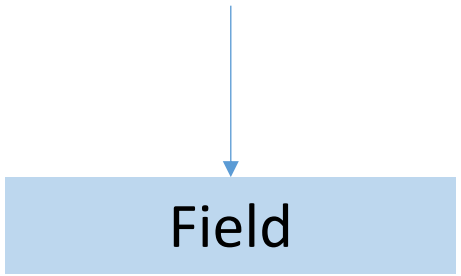
or (not quite the same thing, but for demonstration purposes)

```
db.unicorns.find({gender: {$ne: 'f'}, weight: {$gte: 701}})
```

So this is an AND

- The $exists operator is used for matching the presence or absence of a field, for example:

```
db.unicorns.find({vampires: {$exists: false}})
```



- The $in operator is used for matching one of several values that we pass as an array, for example:

```
db.unicorns.find({loves: {$in:['apple','orange']}})
```

- If we want to OR rather than AND several conditions on different fields, we use the $or operator and assign to it an array of selectors we want or’d:

```
db.unicorns.find({gender: 'f',$or: [{loves: 'apple'},{weight: {$lt: 500}}]})
```

The above will return all female unicorns which either love apples or weigh less than 500 pounds.

- There is something pretty neat going on in our last two examples. You might have already noticed, but the loves field is an array. MongoDB supports arrays as first class objects. This is an incredibly handy feature.
- Once you start using it, you wonder how you ever lived without it.
- What is more interesting is how easy selecting based on an array value is: {loves: 'watermelon'} will return any document where watermelon is a value of loves.

- We have seen how these selectors can be used with the find command. They can also be used with the remove command which we have briefly looked at, the count command, which we have not looked at but you can probably figure out, and the update command which we will spend more time with later on.

- The ObjectId which MongoDB generated for our _id field can be selected like so:

```
db.unicorns.find({_id: ObjectId("TheObjectId")})
```

Example:

```
db.unicorns.find({_id: ObjectId("56b3db6d323460b88657a52a")})
```

ObjectId: Methods and Attributes

- Returns the hexadecimal string representation of the object:

ObjectId("56b3db6d323460b88657a52a").str

- Returns the timestamp portion of the object as a Date:

ObjectId("56b3db6d323460b88657a52a").getTimestamp()

- Returns the JavaScript representation in the form of a string literal "ObjectId(...)":

ObjectId("56b3db6d323460b88657a52a").toString()

- Returns the representation of the object as a hexadecimal string.

The returned string is the str attribute:

- ObjectId("56b3db6d323460b88657a52a").valueOf()

Updating

- We have introduced three of the four CRUD (create, read, update, and delete) operations. Now we focus on update.
- Update has a few surprising behaviors, let us see.
- In its simplest form, update takes two parameters: the selector to use and what updates to apply to fields.
- Suppose that the unicorn “Roooooodles” had gained a bit of weight, you might expect that we should execute:

```
db.unicorns.update({name: 'Roooooodles'}, {weight: 590})
```

- Now, if we look at the updated record:

```
db.unicorns.find({name: 'Roooooodles'})
```

- You should discover the first surprise of update. No document is found! Because the second parameter we supplied did not have any update operators (see next slide $set operator), and therefore it was used to **replace** the original document.

  In other words, the update found a document by name and replaced the entire document with the new document (the second parameter)

```
db.unicorns.find({weight: 590})
```

So if you want to change the value of one, or a few fields, you must use MongoDB's $set operator.
Go ahead and run this update to reset the lost fields:

```
db.unicorns.update({weight: 590}, {$set: {
name: 'Roooooodles',
dob: new Date(1979, 7, 18, 18, 44),
loves: ['apple'],
gender: 'm',
vampires: 99}})
```

Investigar el significado de los parámetros de **Date**(). También ver **ISODate**()

- Therefore, the correct way to have updated the weight in the first place is:

```
db.unicorns.update({name: 'Roooooodles'}, {$set: {weight: 590}})
```

- In addition to $set, there are other operators to do some nifty things. For example, the $inc operator is used to increment a field by a certain positive or negative amount.

Nifty = ingenioso

- If the unicorn “Pilot” was incorrectly awarded a couple vampire kills, we could correct the mistake by executing:

```
db.unicorns.update({name: 'Pilot'}, {$inc: {vampires: -2}})
```

```
db.unicorns.find({name: 'Pilot'})
```

- If Aurora suddenly developed a sweet tooth, we could add a value to her loves field via the $push operator:

```
db.unicorns.update({name: 'Aurora'}, {$push: {loves: 'sugar'}})
```

```
db.unicorns.find({name: 'Aurora'})
```

- The $push operator appends a specified value to an array.
- To remove elements just use $pull

Upserts

- One of the more pleasant surprises of using update is that it fully supports upserts.
- An upsert updates the document if found or inserts it if not.
- To enable upserting we pass a third parameter to update {upsert:**true**}.

- A mundane example is a hit counter for a website.
- If we wanted to keep an aggregate count in real time, we would have to see if the record already existed for the page, and based on that decide to run an update or insert.
- With the upsert option omitted (or set to false), executing the following will not do anything:

```
db.hits.update({page: 'unicorns'}, {$inc: {hits: 1}});
db.hits.find();
```

- However, if we add the upsert option, the results are quite different:

```
db.hits.update({page: 'unicorns'}, {$inc: {hits: 1}}, {upsert:true});
db.hits.find();
```

- Since no documents exists with a field page equal to unicorns, a new document is inserted.
- **If we execute it a second time**, the existing document is updated and hits is incremented to 2:

```
db.hits.update({page: 'unicorns'}, {$inc: {hits: 1}}, {upsert:true});
db.hits.find();
```

- The final surprise update has to offer is that, by default, it will update a single document. So far, for the examples we have looked at, this might seem logical. However, if you executed something like:

```
db.unicorns.update({}, {$set: {vaccinated: true }});
db.unicorns.find({vaccinated: true});
```

You might expect to find all of your precious unicorns to be vaccinated, but only one was vaccinated!

- To get the behavior you desire, the multi option must be set to true:

```
db.unicorns.update({}, {$set: {vaccinated: true }}, {multi:true});
db.unicorns.find({vaccinated: true});
```

More about find

- find could take a second optional parameter called “projection”.
- This parameter is the list of fields we want to retrieve or exclude. For example, we can get all of the unicorns’ names without getting back other fields by executing:

```
db.unicorns.find({}, {name: 1});
```

By default, the _id field is always returned.

We can explicitly exclude it by specifying {name:1, _id: 0}:

```
db.unicorns.find({}, {name:1, _id: 0});
```

- Aside from the _id field, you cannot mix and match inclusion and exclusion. If you think about it, that actually makes sense. You either want to select or exclude one or more fields explicitly.
- **Ordering:** We specify the fields we want to sort on as a JSON document, using 1 for ascending and -1 for descending. For example:

//heaviest unicorns first:

```
db.unicorns.find().sort({weight: -1})
```

//by unicorn name then vampire kills:

```
db.unicorns.find().sort({name: 1, vampires: -1})
```

- **Paging:** Paging results can be accomplished via the limit and skip methods.
- To get the second and third heaviest unicorn, we could do:

```
db.unicorns.find().sort({weight: -1}).skip(1).limit(2)
```

Note that this has the behaviour of a
pipeline process

- The shell makes it possible to execute a count directly on a collection, such as:

```
db.unicorns.count({vampires: {$gt: 50}})
```

Or alternatively:

```
db.unicorns.find({vampires: {$gt: 50}}).count()
```