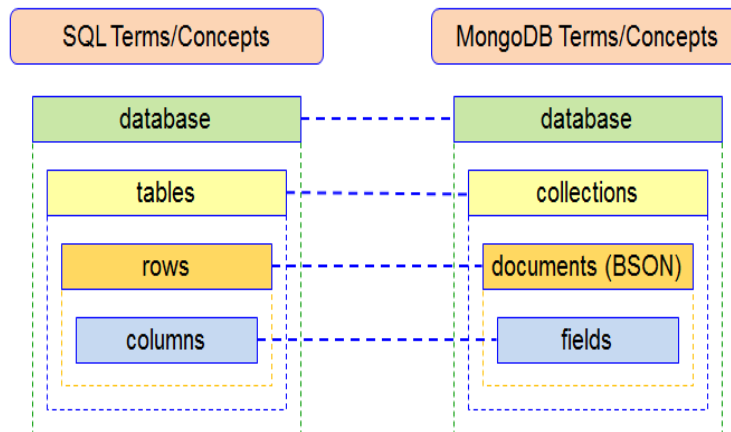


U.T. 4: ACCESO A BASES DE DATOS NOSQL. MONGODB

1. ¿QUÉ ES MONGODB?.
2. ESTRUCTURA DE UNA BASE DE DATOS.
3. PUESTA EN MARCHA DE MONGODB.
4. OTRAS HERRAMIENTAS MONGODB.
5. CONSULTAS.
 - 5.1. Recuperando datos.
 - 5.2. Proyección de campos.
 - 5.3. Modificando documentos.
 - 5.4. Borrando documentos.
6. CONECTAR CON MONGODB.
7. OPERACIONES BÁSICAS EN JAVA.
 - 7.1. Operaciones de creación.
 - 7.2. Operaciones de lectura: búsquedas.
 - 7.3. Operaciones de modificacion.
 - 7.4. Operaciones de borrado.
8. MAPEADO DE CLASES.



1. ¿QUÉ ES MONGODB?

MongoDB es una de las principales representantes de las bases de datos NoSQL. Este acrónimo define aquellas bases de datos que no utilizan el lenguaje SQL para interaccionar con la misma. De forma genérica, se utiliza como forma de identificar aquellas bases de datos que no almacenan la información en forma de tablas relacionales.

Las bases de datos NoSQL aparecen cuando el coste de las unidades de almacenamiento cae en picado a principios de la primera década del siglo XXI, además de la aparición de los desarrollos de software ágiles

que hacen necesario un sistema de almacenamiento de datos que pueda adaptarse y modificarse de forma más sencilla.

Esta facilidad de cambio tiene sus ventajas e inconvenientes: por una parte, al proporcionar un **modelo menos rígido** existe mayor facilidad para tener un modelo de datos dinámico que pueda adaptarse según evoluciona el desarrollo de nuestro software. Pero, por otra parte, la implementación de esquemas menos estrictos puede llegar a condicionar la fiabilidad de los datos almacenados en la misma, recayendo la fiabilidad de los mismos en la pericia de los programadores que desarrollan el software.

MongoDB pertenece a este mundo no relacional y nos permite guardar una gran cantidad de documentos en colecciones (en vez de tablas) de forma distribuida. Entre sus características podemos destacar que MongoDB es “**Schema Less**”: tiene una muy baja rigidez que nos permite construir nuestros documentos con estructuras diferentes sin afectar el funcionamiento de nuestras aplicaciones. Esto no lo podemos hacer con las tablas de las bases de datos relacionales.

Se trata de un sistema gestor de bases de datos orientado a documentos JSON. JSON (acrónimo de *JavaScript Object Notation*, «notación de objeto de JavaScript») es un formato de texto sencillo para el intercambio de datos. Se trata de un subconjunto de la notación literal de objetos de JavaScript, aunque, debido a su amplia adopción como alternativa a XML, se considera (desde el año 2019) un formato independiente del lenguaje. Una de las ventajas de JSON sobre XML es que resulta más sencillo escribir un analizador sintáctico (parser) para él.

En la actualidad, MongoDB se comercializa mediante tres productos:

Mongo Atlas, como plataforma cloud, con una opción gratuita mediante un cluster de 512MB.

MongoDB Community Edition, versión gratuita con versiones para Windows, MacOS y Linux.

MongoDB Enterprise Advanced, versión de pago con soporte, herramientas avanzadas de monitorización y seguridad, y administración automatizada.

Desde <https://www.mongodb.com/try/download/community> podemos descargar la versión Community acorde a nuestro sistema operativo. También para operar con el conjunto de datos que ofrece MongoDB a modo de prueba, podemos volcarlos a nuestro servidor con el siguiente comando:

```
mongorestore --gzip --archive=/home/compartida/ad/ut4/sampleddata.archive.gz
```

2. ESTRUCTURA DE UNA BASE DE DATOS

Antes de seguir debemos aclarar algunos términos de MongoDB:

- **Documentos:** Son la unidad básica de MongoDB. Son los registros dentro de las colecciones donde guardamos nuestra información con el formato BSON, una representación binaria de JSON (Binary JSON). Gracias a este formato tenemos mucha flexibilidad en los tipos de datos que podemos almacenar.
- **Colecciones:** Son agrupaciones de documentos. Son equivalentes a las tablas en las bases de datos relacionales pero NO nos imponen un esquema o estructura rígida para guardar nuestra información.
- **Bases de Datos:** Son los contenedores físicos para nuestras colecciones. Cada base de datos tiene un archivo propio en el sistema de ficheros de nuestra computadora o servidor.
- **Clusters:** Cada cluster puede tener múltiples bases de datos. MongoDB es una base de datos distribuida, lo que nos permite tener una gran escalabilidad horizontal. Esto significa que no sólo podemos aumentar la capacidad de nuestros servidores (memoria, espacio, núcleos, etc), también podemos aumentar la cantidad de servidores para distribuir la carga de nuestras bases de datos.

Con MongoDB es muy fácil empezar nuestros desarrollos y añadir nuevas funcionalidades a medida que se nos van ocurriendo. MongoDB no obliga a diseñar un esquema o modelo antes de poder comenzar a registrar información. Tenemos una gran flexibilidad para modelar o estructurar situaciones de la vida real gracias a la no rigidez de los documentos en formato BSON. Además, tenemos la posibilidad de utilizar documentos embebidos, que no son nada más que documentos dentro de otros documentos. Su uso nos

ayuda a guardar nuestra información en un solo documento. Gracias a ellos evitamos consultar diferentes documentos y colecciones para obtener la información completa.

Pero, por todo esto, las malas prácticas con los documentos embebidos también pueden ser nuestra perdición. Una colección puede tener cientos de documentos con formatos diferentes y sin sentido ni relación entre sí. Por ejemplo, una colección puede tener diferentes documentos, pero cada documento puede tener información diferente. Algunos pueden incluir unos campos tal vez comunes y añadidos otros campos, al mismo tiempo que otros documentos puede que no incluyan esta información. Algunos pueden entender la edad como un número y otros pueden más bien guardar la fecha de nacimiento.

No obstante, **podemos definir un esquema** para validar que los datos que insertamos cumplan restricciones de tipos de datos o elementos que obligatoriamente deben estar rellenos, aunque será materia que no abarcaremos aquí.

Cuanto más grandes sean nuestras aplicaciones y datos, más dificultoso se puede volver el acceso a nuestra información. Puede ser también un auténtico quebradero de cabeza leer nuestros documentos para entender cómo escribir las consultas más básicas. Al final, por ser demasiado fácil puede acabar siendo demasiado complicado. En definitiva, cuando todo se puede, todo puede salir muy mal.

El orden impuesto de SQL no es por nada. Aquí SQL tiene una enorme ventaja porque de ninguna manera nos permite este tipo de desorden. Si comparamos con MongoDB, desobedecer las reglas es mucho más complicado.

Como se ha comentado, *MongoDB* almacena la información y forma el esquema de la base de datos mediante el uso de documentos *JSON*, tal y como se muestra en la siguiente figura:

```
{
  name: "sue",
  age: 26,
  status: "A",
  groups: [ "news", "sports" ]
}
```

← field: value
← field: value
← field: value
← field: value

La notación de objetos utilizada realmente es similar a JSON, se denomina BSON, consistente en almacenar la información en notación JSON con forma binaria. Un ejemplo de este almacenamiento puede ser el siguiente donde se almacena la información de un jugador.

```
{ _id: "20caff50565910ce7d", nombre: "Al Smith", posicion: 1,
  altura: 1.98, peso: 98, estadisticas: { ppp: 28.9, app: 7.6 } }
```

El ejemplo anterior representa un documento que se almacena en la base de datos. Estos documentos se organizan en colecciones que no tienen porqué tener la misma definición ni almacenar el mismo número de campos. En un documento BSON (Binary JSON) podemos almacenar dos tipos de elementos, por una parte objetos y por otra arrays. Los objetos se definen entre llaves sin un nombre asignado y contienen pares etiqueta/valor separados por el carácter ':'. Estos pares etiqueta/valor se definen de forma múltiple separados por una ',' donde las etiquetas pueden ir definidas entre comillas si seguimos el estándar JSON. Los arrays se definen entre corchetes y son listas de valores separados por ','. En el ejemplo siguiente tenemos la definición de un objeto que representa a un equipo que tiene asignado un array con los años en los que ha quedado campeón.

```
{ _id: "f1f77bcf86cd799439011", nombre: "Oakland Oaks", ciudad: "Oakland",
  estadio: "New Stadium", titulos: [1947, 1956, 1975, 2018] }
```

Hay que hacer notar que los objetos se almacenan en MongoDB con un identificador (tipo *ObjectId*) que es un valor de 12 bytes generados de forma que no se produzca valores repetidos. Este campo llamado "_id" puede ser definido también de forma manual y actuará como clave primaria del documento y, solamente en caso de no ser explícitamente definido, se creará de forma automática.

3. PUESTA EN MARCHA DE MONGODB

Una vez descargado [MongoDB](#) o instalado desde los repositorios del sistema operativo, para ponerlo en marcha tendremos que crear la ruta donde queremos que se almacenen nuestras bases de datos. Con ello, lanzaremos el servidor ejecutando el siguiente comando, recordando que, por defecto, *MongoDB* escuchará en el puerto 27017:

```
mongod --dbpath=ruta_base_de_datos
```

Además, *MongoDB* dispone de una consola cliente que podemos lanzar ejecutando el comando `mongo` (o *mongosh*), que intentará conectar directamente con un servidor ubicado en el propio equipo funcionando en el puerto por defecto, el 27017. También podemos especificar esos parámetros desde la línea de comandos al ejecutarlo (junto con usuario y contraseña si fuera necesario también):

```
mongo --host 192.168.100.23 --port 27017 -u usuario -p password
```

Una vez ejecutado entraremos en la consola de *MongoDB* donde, entre otros, tenemos los siguientes comandos disponibles:

- Ver las bases de datos existentes

```
> show dbs
biblioteca    0.203GB
ejemplo      0.203GB
local         0.078GB
```

- Conectar/Crear una base de datos

```
> use biblioteca
switched to db biblioteca
```

- Ver las colecciones dentro de una base de datos

```
> show collections
libros
system.indexes
```

- Eliminar una base de datos (estando en ella)

```
> db.dropDatabase()
{ "dropped" : "biblioteca", "ok" : 1 }
```

- Mostrar ayuda

```
> help
db.help()           help on db methods
db.mycoll.help()    help on collection methods
help admin          administrative help
help connect        connecting to a db help
. . .
```

4. OTRAS HERRAMIENTAS MONGODB

Además del servidor y el cliente, MongoDB ofrece un conjunto de herramientas para interactuar con las bases de datos, permitiendo crear y restaurar copias de seguridad.

Si, por ejemplo, estamos interesados en introducir o exportar una colección de datos mediante JSON, podemos emplear los comandos *mongoimport* y *mongoexport*:

```
mongoimport -d nombreBaseDatos -c coleccion --file nombreFichero.json  
  
mongoexport -d nombreBaseDatos -c coleccion -o nombreFichero.json
```

Otra posibilidad es realizar una copia de seguridad en binario mediante *mongodump*, el cual genera ficheros BSON. Estos archivos posteriormente se restauran mediante *mongorestore*:

```
mongodump -d nombreBaseDatos nombreFichero.bson  
  
mongorestore -d nombreBaseDatos nombreFichero.bson
```

Si necesitamos transformar un fichero BSON a JSON (de binario a texto), tenemos el comando *bsondump*:

```
bsondump file.bson > file.json
```

Para más información sobre copias de seguridad podemos consultar en <https://www.mongodb.com/docs/manual/core/backups/>.

Monitorización. Tanto *mongostat* como *mongotop* permiten visualizar el estado del servidor MongoDB, así como algunas estadísticas sobre su rendimiento. Si trabajamos con MongoAtlas estas herramientas están integradas en las diferentes herramientas de monitorización de la plataforma. A día de hoy, para interactuar y monitorizar, MongoDB tiene su propio IDE conocido como *Mongo Compass*. De una manera flexible e intuitiva, Compass ofrece visualizaciones detalladas de los esquemas, métricas de rendimiento en tiempo real así como herramientas para la creación de consultas.

Existen tres versiones de Compass: una completa con todas las características, una de sólo lectura sin posibilidad de insertar, modificar o eliminar datos (perfecta para analítica de datos) y una última versión *isolated* que solo permite la conexión a una instancia local.

5. CONSULTAS

Como paso previo a las operaciones CRUD puedes hacer uso de alguna web que valide tus pruebas de sintaxis de documentos JSON (búsqueda del tipo: json parser online).

MongoDB proporciona varias vías para ejecutar cada una de las operaciones CRUD (Create, Read, Update, Delete).

<https://www.mongodb.com/docs/manual/crud/>

En MongoDB, el atributo *_id* es único dentro de la colección, y hace la función de clave primaria. Se le asocia un valor *ObjectId*, el cual es un tipo BSON de 12 bytes que se crea mediante:

- la marca de tiempo (*timestamp*) actual (4 bytes)
- un valor aleatorio y único por máquina y proceso (5 bytes)
- un contador inicializado a número aleatorio (3 bytes).

Este objeto lo crea el driver y no MongoDB, por lo cual no deberemos considerar que siguen un orden concreto, ya que clientes diferentes pueden tener timestamps desincronizados. Lo que sí que podemos obtener a partir del *ObjectId* es la fecha de creación del documento, mediante el método *getTimestamp()* del atributo *_id*:

```
> db.people.findOne()._id
< ObjectId( "6316fc938cc2bc168bfed066" )
> db.people.findOne()._id.getTimestamp()
< ISODate( "2022-09-06T07:53:55.000Z" )
```

Este identificador es global, único e inmutable. Esto es, no habrá dos repetidos y una vez un documento tiene un `_id`, éste no se puede modificar. Si en la definición del objeto a insertar no ponemos el atributo identificador, MongoDB creará uno de manera automática. Si lo ponemos nosotros de manera explícita, MongoDB no añadirá ningún `ObjectId`. Eso sí, debemos asegurarnos que sea único (podemos usar números, cadenas, etc.). También, si queremos, podemos hacer que el `_id` de un documento sea un documento en sí, y no un entero, para ello, al insertarlo, podemos asignarle un objeto JSON al atributo identificador.

5.1. Recuperando datos

Para recuperar los datos de una colección o un documento en concreto usaremos el método `find()`. Este método, aplicado sobre una colección, devuelve un cursor a los datos obtenidos, el cual se queda abierto con el servidor y que se cierra automáticamente a los 30 minutos de inactividad o al finalizar su recorrido. Si hay muchos resultados, la consola nos mostrará un subconjunto de los datos (20). Si queremos seguir obteniendo resultados, solo tenemos que introducir `it`, para que continúe iterando el cursor.

MongoDB cuenta con un amplio conjunto de operadores relaciones y lógicos:

| Comparador | Operador |
|----------------------|--------------------|
| menor que | <code>\$lt</code> |
| menor o igual que | <code>\$lte</code> |
| mayor que | <code>\$gt</code> |
| mayor o igual | <code>\$gte</code> |
| igual a | <code>\$eq</code> |
| distinto | <code>\$ne</code> |
| negación | <code>\$not</code> |
| and (conjunción) | <code>\$and</code> |
| or (disyunción) | <code>\$or</code> |
| nor (negación de or) | <code>\$nor</code> |

Al hacer una consulta, si queremos obtener datos mediante más de un criterio, en el primer parámetro del `find` podemos pasar un objeto JSON con los campos a cumplir (condición Y o `and`):

```
db.trips.find({'start station id': 405, 'end station id': 146})
```

Las consultas conjuntivas, es decir, con varios criterios u operador `$and`, deben filtrar el conjunto más pequeño cuanto más pronto posible. Supongamos que vamos a consultar documentos que cumplen los criterios A, B y C. Digamos que el criterio A lo cumplen 40.000 documentos, el B lo hacen 9.000 y el C sólo 200. Si filtramos A, luego B, y finalmente C, el conjunto que trabaja cada criterio es muy grande. En cambio, si hacemos una consulta que primero empiece por el criterio más restrictivo, el resultado con lo que se intersecciona el siguiente criterio es menor, y por tanto, se realizará más rápido.

Estos operadores se pueden utilizar de forma simultánea sobre un mismo campo o sobre diferentes campos, sobre campos anidados o que forman parte de un array, y se colocan como un nuevo documento en el valor del campo a filtrar, compuesto del operador y del valor a comparar mediante la siguiente sintaxis:

```
db.<coleccion>.find( { <campo>: { <operador> : <valor> } } )
```

Por ejemplo, para recuperar los viajes que han durado menos de 5 minutos o comprendidos entre 3 y 5 minutos (el campo almacena el tiempo en segundos), podemos hacer:

```
db.trips.find( { "tripduration" : { $lt : 300 } } )
db.trips.find( { "tripduration" : { $gt : 180 , $lte : 300 } } )
```

Para los campos de texto, además de la comparación directa, podemos usar el operador `$ne` para obtener los documentos cuyo campos no tienen un determinado valor (not equal). Así pues, podemos usarlo para averiguar todos los trayectos realizados por usuarios que no son subscriptores (Subscriber):

```
db.trips.find( { "usertype" : { $ne : "Subscriber" } } )
```

Con cierto parecido a la condición de valor no nulo de las BBDD relacionales y teniendo en cuenta que la libertad de esquema puede provocar que un documento tenga unos campos determinados y otro no lo tenga, podemos utilizar el operador `$exists` si queremos averiguar si un campo existe (y por tanto tiene algún valor):

```
db.people.find( { "edad" : { $exists : true } } )
```

Para usar la conjunción o la disyunción, tenemos los operadores `$and` y `$or`. Son operadores prefijo, de modo que se ponen antes de las subconsultas que se van a evaluar. Estos operadores reciben como parámetro un array, donde cada uno de los elementos es un documento con la condición a evaluar, de modo que se realiza la unión entre estas condiciones, aplicando la lógica asociada a AND y a OR:

```
db.trips.find( { $or : [ { 'start station id' : 405 } , { 'end station id' : 146 } ] } )
db.trips.find( { $or : [ { "tripduration" : { $lte : 70 } } , { "tripduration" : { $gte : 3600 } } ] } )
```

Realmente el operador `$and` no se suele usar porque podemos anidar en la consulta dos criterios, al poner uno dentro del otro. Así pues, estas dos consultas hacen lo mismo:

```
db.trips.find( { 'start station id' : 405 , 'end station id' : 146 } )
db.trips.find( { $and : [ { 'start station id' : 405 } , { 'end station id' : 146 } ] } )
```

Un ejemplo más complejo del uso de los operadores lógicos. Queremos obtener los vuelos (de la colección *routes*) que tienen KZN como origen o destino y avión un CR2 ó A81:

```
db.routes.find( { "$and" : [ { "$or" : [ { "dst_airport" : "KZN" } ,
                                         { "src_airport" : "KZN" }
                                       ] },
                             { "$or" : [ { "airplane" : "CR2" } ,
                                           { "airplane" : "A81" } ] }
               ] } ).pretty()
```

Las consultas disyuntivas, es decir, con varios criterios excluyentes u operador `$or`, deben filtrar el conjunto más grande cuanto más pronto posible. Supongamos que vamos a consultar los mismos documentos que cumplen los criterios A (40.000 documentos), B (9.000 documentos) y C (200 documentos). Si filtramos C, luego B, y finalmente A, el conjunto de documentos que tiene que comprobar MongoDB es muy grande. En cambio, si hacemos una consulta que primero empiece por el criterio menos restrictivo, el conjunto de documentos sobre el cual va a tener que comprobar siguientes criterios va a ser menor, y por tanto, se realizará más rápido. Es justamente a la inversa de como se han de hacer las consultas conjuntivas (operador `$and`). Observa también en la anterior consulta el uso de la función *pretty()*, la cual le da mayor legibilidad a la impresión de resultados en el terminal.

Finalmente, si queremos indicar mediante un array los diferentes valores que puede cumplir un campo, podemos utilizar el operador `$in`:

```
db.trips.find( { "birth year" : { $in : [1977, 1980] } } )
```

Por supuesto, también existe su negación mediante *\$nin*.

Para acceder a campos de subdocumentos, siguiendo la sintaxis de JSON, se utiliza la notación punto. Esta notación permite acceder a cualquier campo de un documento anidado, tal y como sucede en Java al acceder a miembros internos de atributos de tipo objeto (no tipos básicos).

Si trabajamos con arrays, vamos a poder consultar el contenido de una posición del mismo tal como si fuera un campo normal, siempre que sea un campo de primer nivel, es decir, no sea un documento embebido dentro de un array.

```
db.posts.find( { tags : { $in : ["dream", "action"]} } )
```

A los resultados obtenidos con una consulta hecha con *find()* podemos concatenarle llamadas a otras funciones para limitar, ordenar o saltar documentos. Para ello contamos con los siguientes métodos:

| Método | Uso |
|----------------------------|---|
| resultados.limit(cantidad) | Restringe el número de resultados a cantidad |
| resultados.sort({campo:1}) | Ordena los datos por campo: 1 ascendente o -1 o descendente |
| resultados.skip(cantidad) | Permite saltar cantidad elementos con el cursor |

Así pues, si quisiéramos obtener los tres viajes que más han durado, podríamos hacerlo así:

```
db.trips.find().sort( { "tripduration" : -1 } ).limit(3)
```

Desde la versión 4.0, los métodos *count* a nivel de colección y de cursor están caducados (*deprecated*), y no se recomienda su utilización. Aún así, es muy común utilizarlo como método de un cursor:

```
db.trips.find( { "birth year" : 1977, "tripduration" : { $lt : 600 } } ).count()
```

Para contar el número de documentos, en vez de *find* usaremos el método *countDocuments*. Por ejemplo:

```
> db.trips.countDocuments( { "birth year" : 1977 } )
< 186
> db.trips.countDocuments( { "birth year" : 1977, "tripduration" : { $lt : 600 } } )
< 116
```

5.2. Proyección de campos

Las consultas realizadas hasta ahora devuelven los documentos completos. Si queremos que devuelva sólo un campo o varios campos en concreto, hemos de pasar un segundo parámetro de tipo JSON con aquellos campos que deseamos mostrar con el valor *true* o *1*. Hemos de tener en cuenta que si no se indica nada, por defecto siempre mostrará el campo *_id*:

```
> db.trips.find( { 'start station id' : 405, 'end station id' : 146 }, { tripduration : 1 } )
< { _id: ObjectId("572bb8222b288919b68ad191"), tripduration: 1143 }
```

Por lo tanto, si queremos que no se muestre el *_id*, lo podremos a *false* o *0*:

```
> db.trips.find( { 'start station id' : 405, 'end station id' : 146 }, { tripduration : 1, _id:0 } )
< { tripduration: 1143 }
```

Al hacer una proyección, no podemos mezclar campos que se vean (*1*) con los que no (*0*), excepto el campo *_id*.

Si queremos obtener todos los diferentes valores que existen en un campo, utilizaremos el método *distinct* con el nombre del campo como parámetro:

```
> db.trips.distinct( 'usertype' )
< [ 'Customer', 'Subscriber' ]
```

Si queremos filtrar los datos sobre los que se obtienen los valores, le pasaremos un segundo parámetro con el documento que indique el criterio a aplicar:

```
> db.trips.distinct( 'usertype', { "birth year" : { $gt : 1990 } } )
< [ 'Subscriber' ]
```

5.3. Modificando documentos

Vamos a insertar dos veces la misma persona sobre la cual realizaremos las modificaciones:

```
db.people.insertOne({ nombre : "Rafa Nadal", edad : 37, profesion : "Tenista" })
db.people.insertOne({ nombre : "Rafa Nadal", edad : 37, profesion : "Tenista" })
```

Para actualizar (y fusionar datos) utilizamos los métodos *updateOne* / *updateMany*. Ambos métodos requieren 2 parámetros: el primero es la consulta para averiguar sobre qué documentos, y en el segundo parámetro indicamos los campos a modificar, utilizando los operadores de actualización:

```
db.people.updateOne({ nombre : "Rafa Nadal" }, { $set : {nombre : "Rafael Nadal", ingresos: 999999 } })
```

Al realizar la modificación, el terminal nos devolverá información sobre cuántos documentos ha encontrado, modificado y más información. Como hay más de una persona con el mismo nombre, al haber utilizado *updateOne* sólo modificará el primer documento que ha encontrado. Si el criterio de selección no encuentra el documento sobre el que hacer los cambios, no se realiza ninguna acción.

Una precaución a tener en cuenta en versiones antiguas de MongoDB, podíamos pasarle como parámetro de actualización un documento, de manera que MongoDB realizaba un reemplazo de los campos, es decir, si en el origen había 20 campos y en la operación de modificación sólo poníamos 2, el resultado únicamente contendría 2 campos. Es por ello que ahora es obligatorio utilizar los operadores.

Si quisiéramos que, en el caso de no encontrar el documento a modificar, insertase un nuevo documento, acción conocida como *upsert* (*update* + *insert*), habría que pasarle un tercer parámetro al método con el objeto { *upsert* : true }. Si encuentra el documento, lo modificará, pero si no, creará uno nuevo:

```
db.people.updateOne({ nombre : "Rafa Nadal"},
                    { $set : {name : "Rafa Nadal", twitter : "@arafanadal"} },
                    { upsert : true } )
```

MongoDB ofrece un conjunto de operadores para simplificar la modificación de campos. El operador más utilizado es el operador *\$set*, el cual admite los campos que se van a modificar. Si el campo no existe, lo creará. Por ejemplo, para modificar el salario haríamos:

```
db.people.updateOne({ nombre : "Rafa Nadal" }, { $set : { ingresos : 1000000 } })
```

Mediante *\$inc* podemos incrementar el valor de una variable:

```
db.people.updateOne({nombre : "Rafa Nadal"}, { $inc : { ingresos : 1000 } })
```

Para eliminar un campo de un documento, usaremos el operador *\$unset*. De este modo, para eliminar el campo twitter de una persona haríamos:

```
db.people.updateOne({ nombre : "Rafa Nadal"}, { $unset : { twitter : '' } })
```

Otros operadores que podemos utilizar son *\$mul*, *\$min*, *\$max*, *\$currentDate* y *\$rename*.

Para todas estas actualizaciones se ha de tener presente que, si al actualizar la búsqueda devuelve más de un resultado, la actualización sólo se realiza sobre el primer resultado obtenido. Si queremos modificar múltiples documentos, en el tercer parámetro indicaremos `{ multi : true }`:

```
db.grades.update( { type : 'exam' }, { '$inc' : { 'score' : 1 } }, { multi : true } );
```

Esta opción es necesaria únicamente para versiones de MongoDB inferiores a 3.2: `update()` por defecto solo actualiza el primer documento que coincide, pero la realidad actual (MongoDB 5.0+) es que `update()` está marcado como obsoleto. En su lugar se recomienda usar `updateOne()` o `updateMany()`:

```
// Actualiza SOLO el primer documento encontrado
db.grades.updateOne(
  { type: 'exam' },
  { '$inc': { 'score': 1 } }
);

// Actualiza TODOS los documentos que coincidan
db.grades.updateMany(
  { type: 'exam' },
  { '$inc': { 'score': 1 } }
);
```

En resumen, se aconseja `updateOne()` si solo quieres actualizar el primer documento y `updateMany()` si quieres actualizar todos los documentos que coincidan. El viejo `update()` con `{multi: true}` sigue funcionando pero se considerado obsoleto.

También hemos de saber que las actualizaciones múltiples no se realizan de manera atómica ya que MongoDB no soporta transacciones *isolated*. Cada documento sí es atómico, con lo que ninguno se va a quedar a la mitad. Aunque no lo veamos, `findAndModify()` permite encontrar y modificar un documento de manera atómica, evitando que entre la búsqueda y la modificación el estado del documento se vea afectado.

5.4. Borrando documentos

Para borrar usaremos los métodos `remove`, `deleteOne` o `deleteMany`. Si no pasamos ningún parámetro, `deleteOne` borrará el primer documento, o en el caso de `deleteMany` toda la colección documento a documento. Si le pasamos un parámetro, éste será el criterio de selección de los documentos a eliminar:

```
db.people.deleteOne( { nombre : "Selena Williams" } )
```

Al eliminar un documento, no podemos olvidar que cualquier referencia al documento que exista en la base de datos seguirá existiendo. Por este motivo, manualmente también hay que eliminar o modificar esas referencias.

6. CONECTAR CON MONGODB

Para poder trabajar con MongoDB desde cualquier aplicación necesitamos un driver. MongoDB ofrece drivers oficiales para casi todos los lenguajes de programación actuales. Para empezar a trabajar con MongoDB desde Java, primero tendremos que hacernos con el Driver, descargándolo de Internet, de su página web o de los repositorios del sistema operativo (paquete libmongodb-java). Si trabajamos con Maven simplemente añadiremos la dependencia en nuestro `pom.xml`.

```
<dependency>
  <groupId>org.mongodb</groupId>
  <artifactId>mongodb-driver-sync</artifactId>
  <version>5.3.1</version> <!-- Puedes usar la versión más reciente -->
</dependency>
```

También nos será necesaria la documentación [MongoDB JDBC Driver Documentation](#) y de sus clases [MongoDB API Documentation](#).

Para conectar con MongoDB desde Java en versiones anteriores a la 3.7 (con *import com.mongodb.MongoClient*) lo haríamos de la siguiente forma:

```
MongoClient cliente = new MongoClient(); // conecta a localhost, puerto 27017
MongoDatabase db = cliente.getDatabase("basededatos");
```

Podemos especificar explícitamente el nombre del servidor al cual conectamos a su puerto 27017.

```
MongoClient cliente = new MongoClient("localhost");
```

O el nombre y el puerto, imprescindible cuando el servidor funcione en un número de puerto diferente al 27017.

```
MongoClient cliente = new MongoClient("localhost", 27017);
```

A partir de dicha versión 3.7, la conexión se realizará (con *import com.mongodb.client.MongoClient*) con el método estático *MongoClients.create()*, tal y como se explica en

<https://mongodb.github.io/mongo-java-driver/3.7/driver/getting-started/quick-start/>.

```
MongoClient cliente = MongoClients.create("mongodb://localhost:27017");
```

Puedes ver la documentación de la API en

<http://mongodb.github.io/mongo-java-driver/3.7/javadoc/com/mongodb/client/MongoClients.html>.

También podemos conectar haciendo uso de *MongoClientURI*.

```
MongoClientURI connectionString = new MongoClientURI("mongodb://localhost:27017");
MongoClient cliente = MongoClients.create(connectionString);
```

Una vez conectados hemos de crear los objetos asociados a la base de datos y colección correspondientes:

```
MongoDatabase mdb = cliente.getDatabase("biblioteca");
MongoCollection<Document> col = mdb.getCollection("libros");
```

Observa que no es simplemente *MongoCollection*, sino *MongoCollection<Document>*.

Posteriormente trabajaremos con documentos de alguna colección de esa base de datos, y acabaremos desconectando de la Base de Datos.

```
cliente.close();
```

Internamente la implementación de *MongoClient* gestiona un *pool* de conexiones. Lógicamente para poder conectarnos con *MongoDB* el servidor habrá de estar en ejecución.

Mediante el protocolo *mongodb+srv* se establece una conexión segura. Es un protocolo más reciente que el protocolo originario (*mongodb*) y específico para conexiones a *clusters* MongoDB. Este protocolo utiliza registros DNS SRV (registros SRV del protocolo DNS de resolución de nombres de máquina). Los registros SRV permiten que un único nombre de máquina se resuelva en varios nombres, lo que se puede utilizar para descubrir dinámicamente los nodos en un clúster. Esto hace que sea mucho más fácil para los clientes conectarse a clústeres de MongoDB, especialmente para clústeres grandes y distribuidos. En este caso la URL de conexión cambiaría al formato:

```
mongodb+srv://usuario:password@host/basededatos
```

7. OPERACIONES BÁSICAS EN JAVA

Para nuestras aplicaciones Java podremos operar directamente con el driver, es lo que vamos a ver, o utilizar una abstracción de más alto nivel como podría ser, entre otras opciones JPA, Morphia, Hibernate OGM o DataNucleus.

Para trabajar con cada documento vamos a poder operar con las clases *DBObject* o *Document*. La primera de ellas, aunque no está obsoleta (sus métodos *toString()* y *getId()* sí), tiene algunas limitaciones. La segunda, en cambio, simplifica generalmente el código. *DBObject* no se recomienda para aplicaciones nuevas. Es similar a *Document* en que representa los valores BSON, pero tiene algunas deficiencias que fueron imposibles de superar, entre ellas:

- es una interfaz en lugar de una clase, por lo que su API no se puede ampliar sin romper la compatibilidad binaria.
- por ser una interfaz, requiere una clase concreta separada llamada *BasicDBObject* que implemente esa interfaz.

Para unir todo esto, el controlador contiene una interfaz, pequeña pero poderosa, llamada *Bson*. Cualquier clase que represente un documento BSON, ya sea incluida en el propio controlador o de un tercero, puede implementar esta interfaz y luego puede usarse en cualquier lugar de la API de alto nivel donde se requiera un documento BSON.

Para documentos BSON que representan filtros de consulta, especificaciones de actualización, criterios de clasificación, etc., el controlador impone el requisito de que las clases utilizadas para este propósito implementen esta nueva interfaz *Bson*. También *BasicDBObject*, *Document* y *BsonDocument* implementan esa interfaz, por lo que se puede usar cualquiera, o podríamos crear nuestros propios tipos personalizados que lo hagan. El propio controlador devuelve instancias de *Bson* de la mayoría de los métodos estáticos del driver, como los de la clase *Filters*. Un ejemplo de alguno de estos métodos es *com.mongodb.client.model.Filters.eq*.

Regla general:

- usa *Document* para datos/documentos completos
- usa *Bson* (builders estáticos) para filtros, actualizaciones, agregaciones y ordenaciones.

7.1. Operaciones de creación

- *db.collection.insert()*
- *db.collection.insertOne()*
- *db.collection.insertMany()*

Ejemplo desde el cliente en el terminal:

```
db.libros.insert(                                // colección
{                                                  // documento
  titulo: "Secuestrado",
  descripcion: "Aventuras de David Balfour",
  autor: "Robert L. Stevenson",
  fecha: "2/5/2002",
  disponible: true
})
```

En el caso de insertar más de un documento, los incluiríamos en un array (entre signos de corchetes [y]).

El mismo ejemplo anterior desde Java:

```
Libro libro = new Libro("Secuestrado", "Aventuras de David Balfour", "Robert L. Stevenson", "2/5/2002", true);
Document documento = new Document()
    .append("titulo", libro.getTitulo())
    .append("descripcion", libro.getDescripcion())
    .append("autor", libro.getAutor())
    .append("fecha", libro.getFecha())
    .append("disponible", libro.getDisponible());
mdb.getCollection("libros").insertOne(documento);
```

Los constructores de la clase *Document* más utilizados son, primero, el que no admite parámetros (el utilizado en el ejemplo anterior) o, segundo, el que admite dos parámetros: una clave y un valor. Para crear documentos con más parejas clave-valor aplicaremos el método *append()*, que retorna el *Document* modificado, tantas veces como sea necesario encadenando las llamadas con un punto (.) tal y como hemos visto en el anterior ejemplo. Esta será la técnica básica para construir las listas JSON.

En el caso de querer insertar varios al mismo tiempo, utilizaríamos *insertMany*, en lugar de *insertOne*, pasándole como parámetro una colección (*ArrayList*, por ejemplo) de los objetos *Document* correspondientes.

7.2. Operaciones de lectura: búsquedas

Para realizar búsquedas se utiliza el método *find()*. A este método le podemos pasar un documento que haga de patrón para buscar en la colección todos aquellos que coincidan con él, como en los ejemplos vistos para el cliente de Mongo. O también se pueden pasar una serie de métodos que permiten establecer condiciones muy al estilo de las que se pasaban en las cláusulas *WHERE* de las sentencias SQL.

- *find(eq("titulo", "Secuestrado"))* - Buscará todos los documentos que tengan como título el valor 'Secuestrado'. También se puede utilizar para buscar valores dentro de un array en el documento de una colección.
- *find(gt("paginas", 50))* - Buscará todos los documentos que tengan más de 50 páginas.
- *find(gte("paginas", 50))* - Buscará todos los documentos que tengan 50 o más páginas.
- *find(and(gte("paginas", 50), lte("paginas", 100)))* - Buscará todos los documentos que tengan entre 50 y 100 páginas (ambos valores incluidos).
- *find(or(eq("genero", "novela"), eq("genero", "accion")))* - Buscará todos los documentos que sean del género 'novela' o 'accion'.

Los distintos métodos *eq*, *lt*, *gt*, etc. se encuentran definidos dentro de *Filters* como métodos estáticos. Generalmente *Filters* es omitido al hacer uso del *import static* correspondiente.

Ejemplos

- Devuelve el primer documento que coincida con el campo y valor especificados:

```
Document documento = db.getCollection("libros").find( eq( "titulo", "Secuestrado" ) )
    .first();
```

- Devuelve todos los documentos que coincidan con el campo y valor especificados:

```
FindIterable iterable = db.getCollection("libros").find( eq( "titulo", "Secuestrado" ) );
```

Observa que en caso de consulta con múltiples resultados se están retornando del tipo *FindIterable*.

- Devuelve los primeros diez documentos que coincidan con el campo y valor especificados:

```
FindIterable iterable = db.getCollection("libros").find( eq( "titulo", "Secuestrado" ) )
    .limit(10);
```

- Devuelve los documentos encontrados saltándose los diez primeros:

```
FindIterable iterable = db.getCollection("libros").find( eq( "titulo", "Secuestrado" ) )
    .skip(10);
```

- Busca en un documento que tiene otro embebido:

```
FindIterable iterable = mdb.getCollection("libros")
    .find( eq( "editorial.nombre", "Anaya" ) );
```

- Búsqueda sobre dos campos (género y número de páginas). Busca los libros de género *novela* que tengan más de 100 páginas.

```
FindIterable iterable = mdb.getCollection("libros")
    .find( and( eq( "genero", "novela" ), gt( "paginas", 100 ) );
```

Operadores

A continuación, los operadores más habituales:

- Operadores de comparación: *eq*, *gt*, *gte*, *lt*, *lte*, *ne* (*not equal*), *in*, *nin* (*not in*)
- Operadores lógicos: *or*, *and*, *not*

Se pueden encontrar más ejemplos de métodos para realizar búsquedas en el [Tutorial de MongoDB para Java](#).

Las búsquedas o consultas en ocasiones requerirán **proyecciones**, es decir, selección de campos a obtener. Para ello se construye un documento en el que marcamos con valores a uno (1) aquéllos campos que se quieren incluir en la proyección, y, opcionalmente, a cero (0) aquéllos que se pretendan excluir de la misma. En el ejemplo siguiente, desde el cliente en la línea de comandos, sólo interesan el título y la descripción de los 10 primeros libros de Stevenson. Al método *find()* le pasamos dos documentos: el primero el criterio de filtrado y el segundo indica los campos a incluir (sólo indicamos con unos los requeridos, el resto de campos no resulta necesario indicarlos con ceros).

```
db.libros.find(                                // colección
    {autor: {$eq: "Robert Louis Stevenson"}}, // filtrado
    {titulo: 1, descripcion: 1}                // proyección
).limit(10)                                    // modificador del cursor
```

Ejemplo desde Java usando *FindIterable* para recorrer los documentos encontrados.

```
Document documento = new Document("autor", "Robert Louis Stevenson");
FindIterable findIterable = mdb.getCollection("libros")
    .find(documento)
    .projection(fields(excludeId(), include("cod", "title")))
    .limit(10);
// iteramos sobre el FindIterable para guardarlos en un ArrayList
List<Libro> libros = new ArrayList<Libro>();
Libro libro = null;
Iterator<Document> iter = findIterable.iterator();
while (iter.hasNext()) {
    Document documento = iter.next();
    libro = new Libro();
    libro.setId(documento.getObjectId("_id"));
    libro.setTitulo(documento.getString("titulo"));
    libro.setDescripcion(documento.getString("descripcion"));
```

```

    libro.setAutor(documento.getString("autor"));
    libro.setFecha(documento.getDate("fecha"));
    libro.setDisponible(documento.getBoolean("disponible", false));
    libros.add(libro);
}

```

Conviene destacar cómo en las proyecciones si se desea excluir el campo "_id" se ha de hacer de forma explícita, como se puede ver en este ejemplo anterior. En el caso de no especificarlo, dicho campo "_id" será incluido incluso si no se indica.

7.3. Operaciones de modificación

- `db.collection.update()`
- `db.collection.updateOne()`
- `db.collection.updateMany()`
- `db.collection.replaceOne()`

Ejemplo desde consola

```

db.libros.update(                                     // colección
  { titulo: { $eq: "Secuestrado" } },                 // criterio
  { $set: { autor: "Robert Louis Stevenson" } }      // modificación
)

```

Ejemplos desde Java

```

// Modifica un campo específico de un documento
mdb.getCollection("libros").updateOne(new Document("titulo", "Secuestrado"),
    new Document("$set", new Document("autor", "Robert Louis Stevenson")));

// Reemplaza un documento completo
mdb.getCollection("libros").replaceOne(new Document("_id", libro.getId()),
    new Document()
        .append("titulo", libro.getTitulo())
        .append("descripcion", libro.getDescripcion())
        .append("autor", libro.getAutor())
        .append("fecha", libro.getFecha())
        .append("disponible", libro.getDisponible()));

```

7.4. Operaciones de borrado

- `db.collection.remove()`
- `db.collection.deleteOne()`
- `db.collection.deleteMany()`

Ejemplo desde consola

```

db.usuarios.remove(                                   // colección
  { poblacion: "Zaragoza" }                          // criterio
)

```

Ejemplo desde Java:

```
mdb.getCollection("libros").deleteMany(new Document("poblacion", "Zaragoza"));
```

8. MAPEADO DE CLASES

Utilizar CodecRegistry para mapear POJOs a Documentos

BSON, como hemos dicho, es un formato de almacenamiento binario similar a JSON, denominado Binary JSON. Al igual que JSON, admite objetos de documentos incrustados y objetos array, pero BSON tiene algunos tipos de datos que JSON no tiene, como los tipos `Date` y `BinData`.

[Codecs BSON Documentation](#).

La interfaz *Codec* abstrae los procesos de decodificación de un valor BSON en un objeto Java usando un *BsonReader* y de codificación de un objeto Java en un valor BSON usando un *BsonWriter*. El valor BSON puede ser tan simple como un booleano o tan complejo como un documento o un array.

Por su parte, un *CodecRegistry* contiene un conjunto de instancias de *Codec* a las que se accede según las clases de Java que se codifican y decodifican. Las instancias de *CodecRegistry* generalmente se crean a través de métodos estándar estáticos en la clase *CodecRegistries*.

Una vez creado un *CodecRegistry*, ya podemos trabajar directamente con los POJOs para insertar, modificar o leer documentos en la base de datos:

```
import static org.bson.codecs.configuration.CodecRegistries.fromProviders;  
import static org.bson.codecs.configuration.CodecRegistries.fromRegistries;
```

```
ConnectionString connString = new ConnectionString("mongodb://localhost");  
CodecRegistry.pojoCodecRegistry = fromProviders(  
    PojoCodecProvider.builder().automatic(true).build());  
CodecRegistry codecRegistry = fromRegistries(  
    MongoClientSettings.getDefaultCodecRegistry(),.pojoCodecRegistry);  
MongoClientSettings clientSettings = MongoClientSettings.builder()  
    .applyConnectionString(connString)  
    .codecRegistry(codecRegistry)  
    .build();  
  
try(  
    MongoClient mongoClient = MongoClient.create(clientSettings);  
    )  
{  
    MongoDBDatabase mdb = mongoClient.getDatabase("biblio");  
    ...  
}
```

Aquí el método estático *fromRegistries* retorna un *CodecRegistry* que combina la lista de instancias *CodecRegistry* en una única conjunta.

A través del *CodecRegistry* los objetos Java se mapearán automáticamente como documentos a la hora de insertarlos y modificarlos en la colección correspondiente, y también serán mapeados automáticamente a objetos Java desde la colección MongoDB cuando hagamos una lectura. Una observación importante a tener en cuenta es la conveniencia de llamar "id" y no "_id" a la clave. Si la llamamos "_id" inevitablemente, a pesar de que su tipo sea *int*, o *Long*, será creado con el tipo *ObjectId*. Veamos ahora ejemplos de operaciones básicas de inserción, modificación y lectura. Véase <https://docs.spring.io/spring-data/mongodb/docs/1.2.0.RELEASE/reference/html/mapping-chapter.html>.

Si hemos instanciado un *CodecRegistry* con nuestra conexión a MongoDB, suponiendo que tenemos un objeto de la clase *Coche* que queramos insertar como documento en una colección MongoDB, podríamos hacer lo siguiente:

```
MongoCollection<Coche> mco = mdb.getCollection("coches", Coche.class);
```

```
mco.insertOne(coche); // coche es un objeto de la clase Coche
```

Observa en primera lugar que ya no trabajamos con `<Document>`, sino con `<Coche>`. Observa también el segundo parámetro a `getCollection()`. De esa manera obtenemos, mapeando automáticamente a través del `CodecRegistry`, una colección de objetos `Coche` casi directamente de una colección MongoDB. Es decir, la colección ya no lo es de objetos `Document` sino de la clase POJO correspondiente. Para obtener un objeto de la colección será tan fácil como:

```
Libro lc = mco.find(Filters.eq("_id",2)).first();
```

O todos:

```
. . .
MongoCollection<Coche> mco = mdb.getCollection("coches", Coche.class);
List<Coche> coches = mco.find().into(new ArrayList<Coche>());
. . .
```

De nuevo, utilizando `CodecRegistry` para el mapeo automático de los documentos en objetos Java (POJO), podríamos **modificar** un documento de MongoDB pasándole directamente el objeto Java. En el ejemplo siguiente, tenemos el objeto con todos los campos ya modificados (excepto el id, que nunca tocaremos) y, utilizando el `id` como filtro, pasamos el objeto que reemplazará al que haya actualmente en la base de datos:

```
. . .
MongoCollection<Coche> coleccionCoches = db.getCollection("coches", Coche.class);
coleccionCoches.replaceOne(eq("_id", coche.get_id()), coche);
. . .
```

También podemos **borrar** documentos utilizando un `CodecRegistry`. En el ejemplo siguiente, tenemos el objeto cuyo documento queremos eliminar de la base de datos, y utilizamos el `id` como filtro para localizarlo y borrarlo de la colección:

```
. . .
MongoCollection<Coche> coleccionCoches = mdb.getCollection("coches", Coche.class);
coleccionCoches.deleteOne(eq("_id", coche.get_id()));
. . .
```

Puedes ver ejemplos de estas operaciones en <https://mongodb.github.io/mongo-java-driver/3.5/driver/getting-started/quick-start-pojo/>.

Esta documentación, creada por Ricardo Cantó Abad, se distribuye bajo los términos de la licencia Creative Commons Atribución-NoComercial-CompartirIgual 3.0 Unported (CC BY-NC-SA 3.0) (<http://creativecommons.org/licenses/by-nc-sa/3.0/es/deed.ca>).

