Tema 3. Operaciones básicas con Mongo

Tabla de contenidos

3.1 Conceptos básicos 2
3.1.1 Mongo Shell 2
3.1.2 Documentos 2
3.1.3 Colecciones 3
3.1.4 Operaciones básicas 4
3.2 Tipos de datos 5
3.2.1 Tipos de datos básicos 6
3.2.2 Dates 7
3.2.3 Arrays. 7
3.2.4 Documentos embebidos 7
3.2.5 ObjectIds 8
3.3 Insertando y salvando documentos 8
3.4 Eliminando documentos 10
3.5 Actualizando documentos 10
3.5.1 Reemplazamiento 11
3.5.2 Usando modificadores 11

3.1 Conceptos básicos

3.1.1 Mongo Shell

El shell que hemos lanzado con mongo, o mongo.exe levanta un shell de comunicación con el servidor Mongo basado en JavaScript, es decir, podemos ejecutar directamente codigo JavaScript, para ilustrarlo ejecute algunos comandos JavaScript sencillos.

```
> var a=200;
> a
200
> var b=a/5;
> b
40
```

También podemos utilizar librerías JavaScript estándar o definir funciones.

```
> Math.sqrt(b)
6.324555320336759

> new Date("2014/1/10")
ISODate("2014-01-09T23:00:00Z")

> function divBy2(n) { return n/2; }
> divBy2(18)
9
```

No se preocupe si no conoce la sintaxis de JavaScript, en este curso vamos a centrarnos en los comandos de Mongo que permiten interactuar con los datos almacenados en el servidor, y los explicaremos cada uno paso a paso.

3.1.2 Documentos

El corazón de MongoDB es el documento, un conjunto ordenado de claves con valores asociados. Su representación como hemos nombrado en el tema anterior es en JSON, un formato muy intuitivo y que no pensamos que requiera mayor explicación. Este podría ser un ejemplo sencillo de un documento que guarda el nombre, apellidos y dedad de una persona. A la izquierda de los dos puntos el nombre del campo y a la derecha el valor.

```
{
     nombre:”Jose Antonio”,
     apellidos:”Guillem Benedito”,
     edad:35
}
```

Las claves de los documentos:

- No pueden ser nulas.
- No pueden contener los caracteres . (punto) y $ (dólar).
- Puede contener cualquiera de los demás caracteres UTF-8 existentes.
- Son *case-sensitive* (sensible a mayúsculas y minúsculas), por lo que las claves "nombre" y "Nombre" son diferentes, y por tanto consideradas como campos diferentes.
- Las claves dentro de un mismo documento deben ser únicas, no pueden duplicarse. Así por ejemplo el siguiente documento no es válido por tener dos veces la clave nota.

```
{
   nombre:"Jose Antonio",
   nota:8.9,
   nota:7.2
}
```

Por otra parte, los valores de cada clave puede ser cualquiera de los tipos de documento permitidos, que veremos en este mismo tema.

Cada documento en Mongo **debe tener obligatoriamente un campo _id** con valor único y que actuará como identificador único del documento. Es tan necesario este campo que cuando se guarda un documento sin especificarlo, Mongo automáticamente le asigna uno del tipo ObjectId.

```
> db.prueba.save( { a:2} )
WriteResult({ "nInserted" : 1 })
> db.prueba.findOne()
{ "_id" : ObjectId("537f9052c3a7ae2fdcd5182e"), "a" : 2 }
```

Como puede comprobar en el ejemplo anterior, hemos intentado guardar con el comando save un documento, con un solo campo a, con valor 2. Al guardarlo, se ha generado automáticamente un campo "_id" con un valor único.

3.1.3 Colecciones

Una colección es un grupo de documentos, es lo análogo a las tablas en el modelo relacional.

Las colecciones tienen esquemas dinámicos, lo que significa que los documentos dentro de una colección pueden tener múltiples "formas". Por ejemplo, los siguientes documentos podrían guardarse en la misma colección, a pesar de tener diferentes campos, y diferentes tipos de dato.

```
{ nombre:"Jose Antonio", edad:35 }
{ username::"pepito", type:6, active:true }
```

Hay algunas restricciones respecto a nombre que una colección puede tener:

- La cadena vacía ("") no es un nombre válido.

- Lo puede contener el carácter null.
- No se pueden crear colecciones cuyo nombre empiece por “system.”, ya que es un prefijo reservado para colecciones internas.
- No debe contener el carácter $ (dólar).

3.1.4 Operaciones básicas

Con el objetivo de poder conocer los tipos de datos que nos ofrece Mongo con ejemplos, vamos ver de forma rápida algunas operaciones básicas.

Insertar

La función *insert* añade documentos a una colección. Por ejemplo suponga que queremos añadir un comentario a la base de datos de nuestro blog, para lo que definimos inicialmente el objeto post.

```
> post = {"title" : "Mi post", "content" : "Hola mundo", "date" : new
Date()}
{
     "title" : "Mi post",
     "content" : "Hola mundo",
     "date" : ISODate("2014-05-23T18:32:45.652Z")
}
```

Y ahora lo insertamos en una colección de nombre *blog*.

```
> db.blog.insert(post)
WriteResult({ "nInserted" : 1 })
```

Leer

Los comandos *find* y *findOne* se utilizan para consultar una colección. La diferencia es que findOne solamente devuelve un documento, y find, devuelve todos.

De esta forma, podemos por ejemplo recuperar el primer documento insertado en la colección *blog*.

```
> db.blog.findOne()
{
     "_id" : ObjectId("537f94b3d3917f835fea30bd"),
     "title" : "Mi post",
     "content" : "Hola mundo",
     "date" : ISODate("2014-05-23T18:32:45.652Z")
}
```

Actualizar

Si queremos actualizar el post anterior, podemos utilizar el comando *update*, que necesita dos parámetros. El primero es el criterio por el que encontrar el documento a actualizar, y el segundo es el nuevo documento.

Supongamos que ahora añadimos el autor a nuestro post. Lo primero que hacemos es modificar la variable que utilizada antes para guardarlo.

```
> post.author="c3po"
c3po
```

Y ahora ejecutamos update, reemplazando el post con title "Mi post" con el objeto post tras añadirle el author.

```
> db.blog.update({title:"Mi post"}, post)
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

Como output del comando puede ver en *nMatched* el número de documentos que ha hecho mathing con el criterio de actualización (primer parámetro) y en *nModified* el número de documentos modificados con el segundo parámetro.

Si además recuperamos el documento de la base de datos, comprobaremos que el cambio se ha efectuado.

```
> db.blog.findOne()
{
     "_id" : ObjectId("537f94b3d3917f835fea30bd"),
     "title" : "Mi post",
     "content" : "Hola mundo",
     "date" : ISODate("2014-05-23T18:32:45.652Z"),
     "author" : "c3po"
}
```

Borrar

Para borrar un documento de una colección tenemos que especificar el criterio de aquellos a borrar, igual que el primer parámetro del comando *update*.

```
> db.blog.remove({title:"Mi post"})
WriteResult({ "nRemoved" : 1 })
```

3.2 Tipos de datos

Mongo ofrece un amplio rango de tipos de datos en los valores de sus documentos. Veámoslos.

3.2.1 Tipos de datos básicos

null

Null se puede utilizar para representar tanto un valor nulo como un campo que no existe.

```
{ "x" : null }
```

boolean

Tipo booleano que permite los valores *true* y *false*.

```
{ "x" : true }
```

```
{ "y" : false }
```

number

Por defecto Mongo utiliza números el coma flotante (Float). De esta forma los siguientes valores del campo x, son float.

```
{ "x" : 3.14 }
```

```
{ "x" : 3 }
```

Para *integers* o *longs* se tiene que indicar con sus clases propias.

```
{ "x" : NumberInt("3") }
```

```
{ "x" : NumberLong("3") }
```

string

Cualquier cadena de caracteres válidos en UTF-8 se puede representar con el tipo *string*.

```
{ "x" : "hola mundo" }
```

date

Las fechas se guardan en milisegundos desde la época. El *time zone* no se guarda.

```
{ "x" : ISODate("2014-05-25T09:09:17.027Z") }
```

array

Listas de valores se representan como arrays en Mongo.

```
{ "x" : [ 3, 5, 9] }
```

documentos embebidos

Los documentos pueden contener otros documentos embebidos como valor de cualquier campo.

```
{ "x" : { "a":2, "b":90} }
```

object id

El tipo ObjectId es el tipo por defecto para el campo _id, y está diseñado para ser sencillo de generar valores únicos de forma global.

```
{ "_id" : ObjectId("5381b33dd3917f835fea30be") }
```

Veamos con un poco más de detalle algunos de los anteriores tipos.

3.2.2 Dates

Dado que Mongo utiliza el tipo Date de JavaScript, cuando cree un nuevo valor Date, utilice siempre *new Date()*, con el constructor, nunca directamente *Date(…)*, ya que esto no devuelve un objeto Date, solo una representación en string.

3.2.3 Arrays.

Arrays son valores intercambiables que se pueden utilizar tanto para operaciones ordenadas (listas, colas) como para no ordenadas (como conjuntos).

Cada valor de un array puede ser del tipo que sea, no es necesario que todos sean del mismo tipo. Cualquiera de los tipo anteriormente introducidos está permitido, incluso documentos embebidos u otros arrays.

```
{ "cosas" : [ "mesa", 3.5, true }
```

Algo muy importante es que Mongo sabe como trabajar con arrays, y sabe hacer operaciones directamente sobre ellos, como por ejemplo, buscar valores en arrays y crear índices de búsqueda, o actualizaciones atómicas de uno o varios elementos del array.

3.2.4 Documentos embebidos

Documentos se pueden usar como valores para una clave, lo que se denomina documentos embebidos (*embedded document).* Se pueden utilizar para organizar los datos de una forma más natural en vez de una estructura más plana.

Por ejemplo podríamos representar los dirección de una persona con un documento embebido.

```
{
     "nombre":"Jose Márquez",
     "direccion": {
          "calle":"Colon",
          "numero":1,
          "poblacion":"Valencia"
     }
}
```

Con este sencillo ejemplo puede ver como los *embedded documents* pueden cambiar la forma en que trabajamos con los datos. El ejemplo anterior probablemente se habría modelizado en un modelo relacional como dos filas diferentes en dos tablas diferentes, con una clave ajena entre ellas. Cuando se utilizan correctamente, mongo puede ofrecer una representación más natural de la información, pero como contrapartida, puede llevar a un punto de redundancia de datos, es decir, de tener la misma información repetida en diferentes partes.ºº

3.2.5 ObjectIds

La clase ObjectId está especialmente diseñada para ser ligera y fácil de generar en un entorno con múltiples máquinas, propio de la naturaleza distribuida de MongoDB, y es mucho más sencillo de gestionar que por ejemplo un número auto incremental.

ObjectIds utilizan 12 bytes de almacenamiento que se distribuyen de la siguiente forma:

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11
Timestamp | Machine | PID | Increment

Donde:

- Cuatro primero bytes son un timestamp desde la época.
- Los tres siguientes forman un identificador único de la máquina, normalmente un hash de su hostname.
- Para garantizar la unicidad en la misma máquina, en el mismo segundo (timestamp) el PID del proceso que genera el ObjectId también forma parte del ObjectId en los bytes 7 y 8.
- Los últimos tres bytes son un autoincremental para garantizar la unicidad entre dentro del mismo segundo, maquina y proceso.

3.3 Insertando y salvando documentos

Inserts son el método básico para añadir datos a MongoDB. Para insertar un documento en una colección, utilice el método:

```
> db.alumno.insert( {"name":"Antonio Cuenca"})
```

El comando ha añadido automáticamente el campo _id de tipo ObjectId, ya que como hemos explicado, todo documento debe tener un identificador único.

Fíjese también en que en ningún momento hemos tenido que definir previamente ni la existencia, ni la estructura de la nueva colección "alumno", directamente hemos insertado un nuevo documento y todo se ha creado en el mismo paso.

Pero el uso del tipo ObjectId para el campo _id no es obligatorio, podemos utilizar cualquier valor, siempre y cuando garanticemos su unicidad. A continuación insertamos una nueva alumna, especificando que su _id es el número 10 (tipo Long).

```
> db.alumno.insert( { _id:NumberLong(10), name:"Raquel",
apellidos:"Gutierrez Garcia"} )
```

Acto seguido, al recuperar todos los documentos de la colección, podemos comprobar que se ha utilizado el Long 10 como identificador.

```
> db.alumno.find()
{ "_id" : ObjectId("538572582106cb4cc87f8ff3"), "name" : "Antonio
Cuenca" }
{ "_id" : NumberLong(10), "name" : "Raquel", "apellidos" : "Gutierrez
Garcia" }
```

Si ahora intentamos insertar otro alumno con el mismo _id NumberLong(10) fallará, dando error de clave duplicada.

```
> db.alumno.insert( { _id:NumberLong(10), name:"Arantxa",
apellidos:"Pardeza Martin"} )
WriteResult({
     "nInserted" : 0,
     "writeError" : {
          "code" : 11000,
          "errmsg" : "insertDocument :: caused by :: 11000 E11000
duplicate key error index: test.alumno.$_id_  dup key: { : 10 }"
     }
})
```

Batch insert

Si estamos en una situación en la que estamos insertando múltiples documentos, podemos hacer la inserción más rápida utilizando *batch inserts*, que permiten insertar en bloque un array de documentos a la colección. Esto se consigue con solo pasar un array de objetos al comando *insert*.

```
> db.numerosprimos.insert(
[{_id:2},{_id:3},{_id:5},{_id:7},{_id:11},{_id:13},{_id:17},{_id:19},
{_id:23}] )
BulkWriteResult({
     "writeErrors" : [ ],
     "writeConcernErrors" : [ ],
     "nInserted" : 9,
```

```
        "nUpserted" : 0,
        "nMatched" : 0,
        "nModified" : 0,
        "nRemoved" : 0,
        "upserted" : [ ]
})
> db.numerosprimos.find()
{ "_id" : 2 }
{ "_id" : 3 }
{ "_id" : 5 }
{ "_id" : 7 }
{ "_id" : 11 }
{ "_id" : 13 }
{ "_id" : 17 }
{ "_id" : 19 }
{ "_id" : 23 }
```

El batch insert solo puede utilizarse para insertar documentos en una colección a la vez, nunca en varias.

Si la inserción de algún documento falla por cualquier motivo (p.ej clave duplicada), se habrá insertado hasta el documento anterior al erróneo en un orden de izquierda a derecha. El fallido y los posteriores no se insertarán.

3.4 Eliminando documentos

Vaya con cuidado, eliminar colecciones completas es muy sencillo en Mongo. Esto borrará todo, tanto la colección como meta propiedades asociadas a ella o índices creados sobre campos.

```
> db.alumno.drop()
true
```

Por otra parte, para eliminar solo documentos de una colección, tenemos el comando *remove*, que recibe como parámetro el criterio de borrado en forma de documento JSON. En ese caso, solo los documentos que cumplen el criterio se eliminarán de la colección.

```
> db.numerosprimos.remove( {_id:23} )
WriteResult({ "nRemoved" : 1 })
```

3.5 Actualizando documentos

Una vez que el documento está guardado en la base de datos, puede modificarse utilizando el método *update*. Este método recibe dos parámetros, el primero es el criterio de actualización, y el segundo el modificador, que describe los cambios que deben realizarse.

La actualización de un documento es atómica, si dos actualizaciones ocurren en el mismo instante de tiempo, el primero que llegue, sea cual sea, ejecutará sus cambios, y después el segundo, si todavía cumple los criterios.

3.5.1 Reemplazamiento

La forma más simple de hacer un *update* reemplaza completamente un documento con otro nuevo, lo cual puede ser útil en algunos casos. Hagamos la siguiente prueba para comprobarlo. Imaginemos que estamos desarrollando una red social y previamente hemos insertado los datos del usuario con nombre "jose".

```
> db.usuarios.insert({name:"jose",friends:32,enemies:2})
WriteResult({ "nInserted" : 1 })
> db.usuarios.findOne()
{
     "_id" : ObjectId("5386c4fdd73aa9d8f663acda"),
     "name" : "jose",
     "friends" : 32,
     "enemies" : 2
}
```

Si ahora por ejemplo queremos cambiar la estructura del documento, moviendo los campos *friends* y *enemies* dentro de una clave *relationships*, podríamos hacerlo de la siguiente forma.

```
> db.usuarios.update(
... {name:"jose"},
... {name:"jose",relationships: {friends:32,enemies:2}})
> db.usuarios.findOne()
{
     "_id" : ObjectId("5386c4fdd73aa9d8f663acda"),
     "name" : "jose",
     "relationships" : {
          "friends" : 32,
          "enemies" : 2
     }
}
```

Puede comprobar que el primer parámetro que pasamos a *update* es el criterio que deben cumplir aquellos documentos que queremos modificar, que pueden ser uno o muchos. En nuestro caso pedimos aquellos documentos con name igual a "jose", que solo tenemos uno. El segundo parámetro es el nuevo documento completo, que reemplazará al actual, y con la estructura que deseamos.

3.5.2 Usando modificadores

Lo normal no es que necesitemos reemplazar el documento completo, sino que solo tengamos que modificar alguna parte. Esto podemos conseguirlo utilizando modificadores del *update* atomicos (*modifiers),* que son claves especiales que nos permiten especificar operaciones de actualización complejas, como por ejemplo alterar, añadir, eliminar claves, e incluso manipular arrays y documentos embebidos.

3.5.2.1 $set

El modificador $set asigna el valor a un campo. Si el campo todavía no existe en el documento lo creará.

Se utiliza en el segundo parámetro que se le pasa al comando update, esto es, en el modificador. Su sintaxis es:

```
{ $set: { "clave": valor } }
```

Veamos sus posibilidades con un ejemplo. Primero insertamos un nuevo alumno en la colección, únicamente con nombre y apellidos.

```
> db.alumno.insert( { name:"Arturo", apellidos:"Leon Zapata" })

> db.alumno.findOne()
{
     "_id" : ObjectId("538814d8b103f1aadf9217fe"),
     "name" : "Arturo",
     "apellidos" : "Leon Zapata"
}
```

Si ahora queremos almacenar su edad, sin tener que volver a guardar el documento completo con nombre y apellidos, podemos hacerlo con el siguiente comando update y el modificador $set.

```
> db.alumno.update( {name:"Arturo"}, { $set: {edad:17} })
```

Con el primer parámetro del update { name:"Arturo"} identificamos el documento a modificar, y con el segundo parámetro los nuevos valores que queremos poner. Una vez modificado, tenemos el nuevo campo.

```
> db.alumno.findOne()
{
     "_id" : ObjectId("538814d8b103f1aadf9217fe"),
     "name" : "Arturo",
     "apellidos" : "Leon Zapata",
     "edad" : 17
}
```

Podemos darle valor a varios campos a la vez, simplemente informando los pares clave:valor separados por coma. En este ejemplo damos valor a tres campos diferentes con un solo comando.

```
> db.alumno.update( {name:"Arturo"}, { $set:
{nota:8.2,orden:12,actitud:"positiva"} })

> db.alumno.findOne()
{
     "_id" : ObjectId("538814d8b103f1aadf9217fe"),
     "name" : "Arturo",
     "apellidos" : "Leon Zapata",
     "edad" : 17,
     "nota" : 8.2,
```

```
     "orden" : 12,
     "actitud" : "positiva"
}
```

$set puede incluso cambiar el tipo de dato que está modificando. Por ejemplo, con el siguiente comando pasamos de almacenar la nota como un *float*, a almacenar un *array* de *floats* con varias notas del alumno.

```
> db.alumno.update({name:"Arturo"},{$set:{ nota:[8.2,7.8,6.6,9.1]}})
> db.alumno.findOne()
{
     "_id" : ObjectId("538814d8b103f1aadf9217fe"),
     "name" : "Arturo",
     "apellidos" : "Leon Zapata",
     "edad" : 17,
     "nota" : [
          8.2,
          7.8,
          6.6,
          9.1
     ],
     "orden" : 12,
     "actitud" : "positiva"
}
```

O añadir como un nuevo campo otro documento embebido, como en este caso, la dirección.

```
> db.alumno.update({name:"Arturo"},{$set:{direccion:{
calle:"Timoneda", numero:6, cp:"46015" }}})
> db.alumno.findOne()
{
     "_id" : ObjectId("538814d8b103f1aadf9217fe"),
     "name" : "Arturo",
     "apellidos" : "Leon Zapata",
     "edad" : 17,
     "nota" : [
          8.2,
          7.8,
          6.6,
          9.1
     ],
     "orden" : 12,
     "actitud" : "positiva",
     "direccion" : {
          "calle" : "Timoneda",
          "numero" : 6,
          "cp" : "46015"
     }
}
```

Con el modificador *$set* podemos alcanzar y modificar cualquier valor dentro de la estructura del documento, sean campos que cuelgan directamente del "raíz" o embebidos en otros campos.

A continuación, el comando que utilizaríamos por ejemplo para modificar el código postal de la dirección del alumno, que se encuentra en el campo "cp" del campo "direccion".

```
> db.alumno.update({name:"Arturo"},{$set:{ "direccion.cp" : "46020"
}})
> db.alumno.findOne()
{
     "_id" : ObjectId("538814d8b103f1aadf9217fe"),
     "name" : "Arturo",
     "apellidos" : "Leon Zapata",
     "edad" : 17,
     "nota" : [
          8.2,
          7.8,
          6.6,
          9.1
     ],
     "orden" : 12,
     "actitud" : "positiva",
     "direccion" : {
          "calle" : "Timoneda",
          "numero" : 6,
          "cp" : "46020"
     }
}
```

3.5.2.2 $unset

Para eliminar cualquier campo de uno o varios documentos lo hacemos también con el comando *update* pero con el modificador *$unset*. Su sintaxis es:

```
{ $unset: { "clave": 1 } }
```

En MongoDB es habitual utilizar los valores 1 y -1 para indicar verdadero y falso respectivamente. En este caso, al especificar valor 1 estamos diciendo que la clave entra dentro del conjunto de campos en los que queremos aplicar el $unset.

De esta forma, si queremos eliminar el campo "actitud" del documento que guarda la información del alumno "Arturo", lo haríamos así.

```
> db.alumno.update({name:"Arturo"},{$unset:{actitud:1}})
> db.alumno.findOne()
{
     "_id" : ObjectId("538814d8b103f1aadf9217fe"),
     "name" : "Arturo",
     "apellidos" : "Leon Zapata",
     "edad" : 17,
     "nota" : [
          8.2,
          7.8,
          6.6,
          9.1
     ],
     "orden" : 12,
     "direccion" : {
          "calle" : "Timoneda",
          "numero" : 6,
          "cp" : "46020"
     }
}
```

3.5.2.3 $inc

El modificador $*inc* puede utilizarse para incrementar o decrementar el valor numérico de una clave existente o para crear una nueva si no existe. Es muy útil para actualizar estadísticas, votos, visitas, hits, etc.

Supongamos que queremos llevar actualizado un campo "puntos" del alumno, donde dependiendo de su comportamiento, tareas, actitud, etc, vamos sumando o restando a dicho campo. Inicialmente, a nuestro alumno de prueba "Arturo", le asignamos una puntuación de 2, para ello utilizamos directamente el modificador $inc, ya que aunque el campo no exista, tomará un valor por defecto de cero y le sumará 2.

```
> db.alumno.update({name:"Arturo"},{$inc:{puntuacion:2}})
> db.alumno.findOne()
{
     "_id" : ObjectId("538814d8b103f1aadf9217fe"),
     "name" : "Arturo",
     "apellidos" : "Leon Zapata",
     "edad" : 17,
     "nota" : [
          8.2,
          7.8,
          6.6,
          9.1
     ],
     "orden" : 12,
     "direccion" : {
          "calle" : "Timoneda",
          "numero" : 6,
          "cp" : "46020"
     },
     "puntuacion" : 2
}
```

Ahora, nuestro alumno realiza unas prácticas voluntarias, e incrementamos en 5 su puntuación, con lo que pasa a tener un valor de 7 (2+5).

```
> db.alumno.update({name:"Arturo"},{$inc:{puntuacion:5}})
> db.alumno.findOne()
{
     "_id" : ObjectId("538814d8b103f1aadf9217fe"),
     "name" : "Arturo",
     "apellidos" : "Leon Zapata",
     "edad" : 17,
     …
     "puntuacion" : 7
}
```

El modificador se comporta exactamente igual para decrementar, solo hay que pasarle un valor negativo con la cantidad a restar. En el ejemplo restamos 3 al valor almacenado de 7.

```
> db.alumno.update({name:"Arturo"},{$inc:{puntuacion:-3}})
> db.alumno.findOne()
{
     "_id" : ObjectId("538814d8b103f1aadf9217fe"),
```

```
    "name" : "Arturo",
    "apellidos" : "Leon Zapata",
    "edad" : 17,
    …
    "puntuacion" : 4
}
```

3.5.2.4 Arrays: $push

Existen una extensa clase de modificadores para manipular arrays en mongo. El primero que veremos es *push*, que se utiliza para añadir elementos a un array. Si el array no existe lo crea con los elementos indicados en el *push*, y si ya existe los añade al final del array.

Supongamos que queremos ir guardando en la propia colección de alumno, en cada documento que representa un alumno, sus notas en las diferentes asignaturas que está matriculado, esto podríamos conseguirlo cómodamente con un array, veamos como:

Para empezar, y para que el ejemplo resulte más ilustrativo, vamos a borrar la colección alumno.

```
> db.alumno.drop()
true
```

Y ahora, insertamos un nuevo alumno con los datos que queramos.

```
> db.alumno.insert({name:"Sofia", apellidos:"Alarcon Sevilla"})
> db.alumno.findOne()
{
    "_id" : ObjectId("538eaefd5cf96f7d9b8440b0"),
    "name" : "Sofia",
    "apellidos" : "Alarcon Sevilla"
}
```

Para añadir asignaturas con sus notas utilizamos $push cuya sintaxis es:

```
    { $push: { "clave": elemento } }
```

Donde elemento es un objeto JSON que será añadido al campo “clave”. De esta forma para añadir a la clave asignaturas, que Sofía ha sacado un 9.1 en matemáticas, sería así:

```
> db.alumno.update({name:"Sofia"},{$push:{"asignaturas":
{name:"Matematicas", nota:9.1}}})
> db.alumno.findOne()
{
    "_id" : ObjectId("538eaefd5cf96f7d9b8440b0"),
    "name" : "Sofia",
    "apellidos" : "Alarcon Sevilla",
    "asignaturas" : [
        {
            "name" : "Matematicas",
            "nota" : 9.1
        }
    ]
}
```

Puede comprobar que el array “asignaturas” se ha creado con el propio comando update, conteniendo el único elemento que le hemos pasado. Si ahora queremos añadir otra nota, el procedimiento es exactamente el mismo.

```
> db.alumno.update({name:"Sofia"},{$push:{"asignaturas":
{name:"Lenguaje", nota:8.5}}})
> db.alumno.findOne()
{
     "_id" : ObjectId("538eaefd5cf96f7d9b8440b0"),
     "name" : "Sofia",
     "apellidos" : "Alarcon Sevilla",
     "asignaturas" : [
          {
               "name" : "Matematicas",
               "nota" : 9.1
          },
          {
               "name" : "Lenguaje",
               "nota" : 8.5
          }
     ]
}
```

3.5.2.5 Arrays: $pull

Hay varias formas de eliminar elementos de un array. Cuando queremos borrar elementos basados en algún criterio, el modificador adecuado es *$pull.*

Por ejemplo, supongamos que tenemos una lista de tareas que tenemos que hacer, sin un orden específico:

```
> db.lists.insert({"todo": ["lavar platos", "colada", "tender"]})
> db.lists.findOne()
{
     "_id" : ObjectId("53932b7a07db5649da958361"),
     "todo" : [
          "lavar platos",
          "colada",
          "tender"
     ]
}
```

Y ahora, conforme vamos haciendo tareas, las vamos eliminando del array. Por ejemplo para eliminar la colada haríamos.

```
> db.lists.update({},{$pull:{"todo":"colada"}})
> db.lists.findOne()
{
     "_id" : ObjectId("53932b7a07db5649da958361"),
     "todo" : [
          "lavar platos",
          "tender"
     ]
}
```

Al hacer $pull eliminamos todos los elementos que hacen matching, no solo uno. Si por ejemplo tuviéramos un array [1,1,2,1] e hiciéramos $pull del valor 1, finalmente quedaría un array de esta forma [2].

Existen otras formas de eliminar elementos de un array, si por ejemplo consideramos el array como una cola. Para ver un ejemplo, añadamos primero otros ítems a la lista de tareas.

```
> db.lists.update({},{$push:{todo:"hacer la compra"}})
> db.lists.update({},{$push:{todo:"barrer"}})
> db.lists.findOne()
{
     "_id" : ObjectId("53932cc607db5649da958362"),
     "name" : "Jorge",
     "todo" : [
          "lavar platos",
          "tender",
          "hacer la compra",
          "barrer"
     ]
}
```

A continuación, eliminamos con {$pop:{todo:1}} el último elemento de la cola. Puede comprobar que borramos el item “barrer”.

```
> db.lists.update({},{$pop:{todo:1}})
> db.lists.findOne()
{
     "_id" : ObjectId("53932cc607db5649da958362"),
     "name" : "Jorge",
     "todo" : [
          "lavar platos",
          "tender",
          "hacer la compra"
     ]
}
```

Si queremos sacar el primer elemento de la lista, también lo hacemos con $pop, pero con el valor -1 en vez de valor 1, esto es {$pop:{todo:-1}}. Puede comprobar que borramos el item “lavar platos”.

```
> db.lists.update({},{$pop:{todo:-1}})
> db.lists.findOne()
{
     "_id" : ObjectId("53932cc607db5649da958362"),
     "name" : "Jorge",
     "todo" : [
          "tender",
          "hacer la compra"
     ]
}
```

Por último, también podemos modificar los elementos de un array por posición, siendo la primera posición el índice cero. Así si por ejemplo tenemos el array de asignaturas de la alumna Sofia:

```
> db.alumno.findOne({name:"Sofia"})
{
     "_id" : ObjectId("538eaefd5cf96f7d9b8440b0"),
     "name" : "Sofia",
     "apellidos" : "Alarcon Sevilla",
```

```
     "asignaturas" : [
          {
               "name" : "Matematicas",
               "nota" : 9.1
          },
          {
               "name" : "Lenguaje",
               "nota" : 8.5
          }
     ]
}
```

Y queremos modificar la nota de su examen de Lenguaje, que está en la segunda posición (índice=1), lo haríamos poniendo el número del indice justo después del nombre del array, es decir "asignaturas.1.", si quisieramos la tercera posición sería "asignaturas.2", etc, con eso estamos accediendo directamente al documento en esa posición del array.

```
> db.alumno.update({name:"Sofia"},{$set:{"asignaturas.1.nota":8.9}})
> db.alumno.findOne()
{
     "_id" : ObjectId("538eaefd5cf96f7d9b8440b0"),
     "name" : "Sofia",
     "apellidos" : "Alarcon Sevilla",
     "asignaturas" : [
          {
               "name" : "Matematicas",
               "nota" : 9.1
          },
          {
               "name" : "Lenguaje",
               "nota" : 8.9
          }
     ]
}
```

3.5.2.6 Upserts

Un *upsert* es un tipo de update espcial. Si no se encuentra ningún documento que haga *matching* con el criterio del update, entonces se creará un nuevo documento combinando el criterio y lo que se quiere actualizar. Si se encuentra un documento que haga *matching* se actualizará normalmente. *Upserts* pueden ser muy útiles porque pueden eliminar la necesidad de buscar en la colección, para saber si debe insertarse o actualizarse.

Para decirle a MongoDB que queremos hacer un upsert, solo hay que pasar al comando *update* un tercer parámetro, con valor true. Esto significa que el update se comportará como hemos explicado que lo hace el upsert.

```
update({…},{…},true)
```

Para comprobar el funcionamiento, primero actualizaremos los apellidos de la alumna con name Sofía, y veremos que se actualiza el documento que hace *matching* con el criterio del update.

```
> db.alumno.update({name:"Sofia"},{$set:{apellidos:"Alarcon
Revilla"}},true)
> db.alumno.findOne()
{
     "_id" : ObjectId("538eaefd5cf96f7d9b8440b0"),
     "name" : "Sofia",
     "apellidos" : "Alarcon Revilla",
     "asignaturas" : [
          {
               "name" : "Matematicas",
               "nota" : 9.1
          },
          {
               "name" : "Lenguaje",
               "nota" : 8.9
          }
     ]
}
```

Como el documento con name:"Sofia" existe, se ha actualizado el campo apellidos. Si ahora probamos el mismo update con un criterio para el que no haya documentos, se insertará un nuevo documento combinando el criterio y los datos que queríamos actualizar. Sino hubiéramos hecho un *upsert* mediante el último parámetro, no se hubiera hecho nada al no haber ningún matching.

```
> db.alumno.update({name:"Maria"},{$set:{apellidos:"Cubero
Alandes"}},true)

> db.alumno.findOne({name:"Maria"})
{
     "_id" : ObjectId("5394a1f5b4421eafa6edbab1"),
     "name" : "Maria",
     "apellidos" : "Cubero Alandes"
}
```

3.5.2.7 Save

El comando *update* es muy útil cuando queremos hacer modificaciones en base a criterios a cumplir, pero no tanto cuando tenemos un documento en JSON y queremos actualizarlo si existe o insertarlo si no. En este caso el comando *save* llega a nuestro rescate.

En este ejemplo lo primero que hacemos es volcar sobre una variable x el documento de la alumna "Maria".

```
> var x = db.alumno.findOne({name:"Maria"})
> x
{
     "_id" : ObjectId("5394a1f5b4421eafa6edbab1"),
     "name" : "Maria",
     "apellidos" : "Cubero Alandes"
}
```

A continuación añadimos a la variable x un nuevo campo edad, con valor 17.

```
> x.edad=17
17
```

Y guardamos el documento con save. Como el objeto x tiene un campo _id, y ese _id existe en la colección alumno, se actualiza el documento de la alumna Maria.

```
> db.alumno.save(x)
> db.alumno.findOne({name:"Maria"})
{
     "_id" : ObjectId("5394a1f5b4421eafa6edbab1"),
     "name" : "Maria",
     "apellidos" : "Cubero Alandes",
     "edad" : 17
}
```

Si ahora hacemos lo mismo, pero con un documento que no existe en la colección, lo insertará.

```
> var y = { name:"Antonio",apellidos:"Gorriz Mendez"}
> y
{ "name" : "Antonio", "apellidos" : "Gorriz Mendez" }
> db.alumno.save(y)
WriteResult({ "nInserted" : 1 })
```