Operaciones CRUD en MongoDB

Las operaciones CRUD son las operaciones básicas que se pueden realizar sobre una base de datos. CRUD es el acrónimo de *Create, Read, Update, Delete*.

En este documento veremos cómo se realizan estas operaciones en MongoDB desde la *shell* de mongo (`mongosh`). Empezaremos por la creación de documentos. A continuación veremos lecturas / consultas para ver cómo se definen y usan los selectores, ya que se usarán los selectores en el resto de operaciones. Y finalmente iremos viendo el resto de operaciones: modificar (*update*) y borrar (*delete*).

Create / Insertar `insert`

Para crear documentos en una colección usaremos los comandos `insertOne` o `insertMany`.

`db.<nombre de la colección>.insertOne(<json del documento>, <documento de opciones>)`

`db.<nombre de la colección>.insertMany(<array con los documentos json>, <documento de opciones>)`

Los métodos `insertOne` e `insertAny` también aceptan un argumento con opciones.

El campo `_id`

Si el documento no especifica un campo `_id`, entonces MongoDB añade el campo `_id` y le asigna un `ObjectId()` único para el documento. La mayoría de los drivers crean un `ObjectId` y lo asignarán al atributo `_id`, pero el `mongod` creará y rellenará el campo `_id` si el driver o la aplicación no lo hace.

Opciones `writeConcern` y `ordered`

Como segundo argumento de `isertOne` e `insertMany` podemos pasar un documento JSON con opciones.

El método `insertOne` admitirá únicamente la opción `writeConcern` mientras que `insertMany` admitirá las opciones `writeConcern` y `ordered`.

El documento de opciones tiene la siguiente estructura:

```
{
    writeConcern: <documento JSON>,
    ordered: <boolean>
}
```

Opción `writeConcern`

Esta opción indica el nivel de *write concern* para la operación. El *write concern* es un mecanismo que permite exigirle a MongoDB que extienda la escritura a una serie de nodos réplica para que la operación se pueda considerar correcta. El *write concern* se especifica mediante un documento JSON con los siguientes campos:

```
{
    w: <valor>,
    j: <boolean>,
    wtimeout: <valor>
}
```

- La opción `w` pide que se confirme que la operación de escritura se ha propagado a un número determinado de nodos réplica. El valor de `w` puede ser un número entero o la cadena `majority`. Si se especifica un número entero, la operación de escritura se confirma una vez que se ha propagado a ese número de nodos réplica. Si se especifica `majority`, la operación de escritura se confirma una vez que se ha propagado a la mayoría de los nodos réplica del clúster. El valor por defecto es `1`.
- La opción `j` pide que se confirme que la operación de escritura se ha escrito en el diario del disco (*on-disk journal*) de MongoDB. El valor por defecto es `false`.
- La opción `wtimeout` especifica un límite de tiempo para la confirmación de la operación de escritura. Si la operación de escritura no se ha confirmado en el tiempo especificado, la operación fallará. El valor por defecto es `0`, que significa que no hay límite de tiempo (con lo que la operación quedaría bloqueada hasta que se confirme la escritura).

Opción `ordered`

Esta opción sólo se puede usar con `insertMany`. Si se especifica como `true` (valor por defecto) las operaciones de inserción se ejecutarán en orden y se detendrán en la primera operación que falle. Si se especifica como `false` las operaciones de inserción se ejecutarán en paralelo y se ignorarán los errores de inserción.

Read / Consultar `find`

Este función sirve para realizar consultas sobre las colecciones de la base de datos. Para *filtrar* o *seleccionar* los datos que queremos obtener habrá que pasar como argumento cierta información que indique qué datos queremos seleccionar. Esta información se pasará en forma de documento JSON que recibe el nombre de *query document*.

Iremos viendo estos selectores poco a poco, ya que también se usarán en las operaciones de modificación y borrado.

Para consultar los documentos de una colección que cumplan una determinada condición se usa el método `find()` y, en caso de que sólo nos interese el primer documento que cumpla la condición podemos usar `findOne()`.

```
db.<nombre de la colección>.find(<documento consulta>, <documento de proyección>,
<documento de opciones>)
```

Documento de consulta

El documento de consulta es un documento JSON que indica qué documentos de la colección queremos obtener. Este documento se pasa como primer argumento de la función `find()`.

Un ejemplo de documento de consulta sería:

```
{
    nombre: 'Manuel',
    apellidos: 'Piñeiro'
}
```

Este documento indica que queremos obtener los documentos de la colección que tengan el campo `nombre` con el valor `Manuel` y el campo `apellidos` con el valor `Piñeiro`.

En los siguientes apartados veremos cómo se pueden construir estos documentos de consulta en más detalle.

Documento de proyección

El documento de proyección es, de nuevo, un documento JSON que indica qué campos de los documentos queremos obtener. Este documento se pasa como segundo argumento de la función `find()`.

Un ejemplo de documento de proyección sería:

```
{
    nombre: 1,
    apellidos: 1,
    notas: 1
}
```

Este documento indica que queremos obtener los campos `nombre`, `apellidos` y `notas` de los documentos seleccionados. El valor `1` indica que queremos obtener el campo y el valor `0` indica que no queremos obtenerlo.

Documento de opciones

El documento de opciones es un documento JSON que indica ciertas opciones para la consulta. Este documento se pasa como tercer argumento de la función `find()`.

Un ejemplo de documento de opciones sería:

```
{
    limit: 10,
    skip: 5,
    sort: { nombre: 1, apellidos: 1 }
}
```

Este documento indica que queremos obtener como máximo 10 documentos, que nos saltaremos los 5 primeros y que queremos que los resultados estén ordenados por el campo `nombre` y, en caso de empate, por el campo `apellidos` de manera creciente.

En este enlace se puede consultar la lista completa de opciones. Como se puede ver, el número de opciones es muy grande y no las veremos en este documento.

Cursores

El método `find()` **no devuelve los documentos buscados** si no que devuelve un **cursor** a los mismos. Un cursor es un objeto que contiene referencias a los documentos seleccionados permite acceder a ellos recorriendo o *iterando sobre el cursor*. *Recorrer* el cursor es una operación que *consume* el cursor, es decir, que una vez que se ha recorrido el cursor ya no se puede volver a recorrer.

Por ejemplo, si obtenemos los 10 primeros elementos de una colección con el método `find()` y luego procesamos el cursor resultante:

```
test> use airbnb_bar

airbnb_bar> let listings = db.listings.find( { }, { }, { limit: 10 } )

airbnb_bar> let c = 0

airbnb_bar> listings.forEach( doc => print(++c) )
1
2
3
4
5
6
7
8
9
10

airbnb_bar> listings.hasNext()
false
airbnb_bar> listings.next()
MongoCursorExhaustedError: Cursor is exhausted
```

El cursor **se mantiene en el servidor** de MongoDB y tiene una vida limitada. Si no se recorre el cursor en un tiempo determinado, MongoDB lo cerrará automáticamente.

Cuando hacemos una consulta en la *shell* de mongo y **no asignamos** el resultado a una variable, la consola de mongo itera por su cuenta sobre el cursor y muestra los resultados en pantalla. Estos resultados se muestran en bloques de 20 documentos.

Nota: No hay una forma simple de conocer el número de resultados que devuelve un cursor **sin consumir** el mismo. Tanto el método `count()` (no recomendado y no disponible en algunos cursores) como el método `explain` **consumirán el cursor**.

Nota: Para conocer el número de resultados que, probablemente, vamos a obtener con una consulta hemos de usar el método `countDocuments(<query document>, <options>)`. Para conocer el número de documentos de una colección podemos usar el método `estimatedDocumentCount()`.

Existen varias formas para iterar sobre un cursor en la shell de mongo:

Método `next()`

```
var cursor = db.airbnb_bar.find()

while (cursor.hasNext()) {
    printjson(cursor.next());
}
```

Método `forEach()`

```
db.arirbnb_bar.find().forEach(function(doc) {
    printjson(doc);
});
```

o lo que es lo mismo:

```
db.airbnb_bar.find().forEach(printjson)
```

Método `toArray()`

Podemos convertir un cursor a un array con el método `toArray()`. Hay que tener en cuenta que este método puede consumir mucha memoria si el cursor contiene muchos documentos ya que carga en la RAM **todos los documentos del cursor**.

```
var cursor = db.airbnb_bar.find()
let array = cursor.toArray()

printjson(array[6])
```

Consultas con una condición

Es el tipo más simple de consulta.

Para hacer un filtro con una única condición simplemente se incluye un JSON con el **atributo** que queremos seleccionar y el **valor** del mismo que nos interesa. Si queremos seleccionar los documentos de la colección **usuarios** que están activos podríamos de filtro:

```
{
    'activo': true
}
```

pasándole este filtro a la función `find()` quedaría:

```
db.usuarios.find( { activo: true } )
```

(`find()` o `find({})`, sin filtro (o con un filtro *vacío*), devolverá (un cursor a) todos los documentos de la colección).

Por ejemplo, para obtener los documentos de la colección `airbnb_bar` cuyo campo `host_neighbourhood` sea `Dreta de l'Eixample`, escribiríamos:

```
db.det_listings.find( { host_neighbourhood: "Dreta de l'Eixample" } )
```

Antes de continuar con selectores más complejos veamos un par de métodos que nos pueden ser útiles para depurar nuestras consultas.

Contar resultados

Para contar los resultados de una consulta se puede usar el método `count()`.

```
db.<nombre de la colección>.find(<filtro>).count()
```

este método está obsoleto y no está disponible en algunos cursores. En su lugar se recomienda usar `countDocuments()`.

```
db.<nombre de la colección>.countDocuments(<query>)
```

Limitar el número de resultados

Para limitar el número de valores que obtendremos como resultado de una consulta tenemos la función `limit()`. Aceptará como argumento un número entero en el que le indicamos cuantas respuestas nos interesan.

```
db.<nombre de la colección>.find(<filtro>).limit(<número de resultados>)
```

`limit()` se usa frecuentemente en combinación con `skip()` y `sort()`:

- `skip(<número>)`: Sirve para *saltarse* cierto número de documentos en un resultado (cursor).
- `sort(<documento>)`: Sirve para ordenar los resultados en función a uno o más campos.

Ejemplo de sort:

En el documento que se le pasa al sort se indican los campos sobre los que se quiere realizar la ordenación (numérica o lexicográfica). El valor del campo será `1` o `-1`, para indicar si queremos que la ordenación sea creciente o decreciente.

Con este comando obtendremos los tres alumnos con mejor nota (en caso de empate se ordenarán alfabéticamente por nombre y apellidos).

```
db.alumnos.find({aprobado: true}).sort({nota: -1, nombre: 1, apellidos:
1}).limit(3)
```

Consulta con múltiples condiciones

Este selector es igual al anterior pero incluyendo en el JSON todos los pares **atributo - valor** que nos interese. Es equivalente a un **and** lógico.

```
{
    'rol': 'admin',
    'activo': true
}
```

sería equivalente a:

```
{
    $and: [ { 'rol': 'admin' }, { 'activo': true } ]
}
```

Consulta con múltiples condiciones válidas (`$or`)

Para realizar este tipo de consulta hemos de incluir un *atributo*, `$or`, cuyo valor ha de ser un **array** con las condiciones que han de cumplir los documentos que nos interesan. Obviamente los documentos seleccionados han de cumplir **al menos una** de las condiciones. Es equivalente a un **or** lógico.

El siguiente código nos devolvería todos los usuarios que sean administradores **o que** estén activos.

```
{
    $or: [ { 'rol': 'admin' }, { 'activo': true } ]
}
```

Consulta con selección por exclusión (`$not`)

Normalmente lo usaremos junto con el atributo `$eq` para significar *not equal*, es decir, distinto.

El siguiente selector nos devolvería todos los usuarios cuyo nombre **no sea** `Manuel`.

```
{
    'nombre': { $not: { $eq: "Manuel" } }
}
```

Operador `$exists`

Este operador se usa para seleccionar los documentos en los que exista un campo. El valor del campo puede ser `true` o `false` para indicar si el campo ha de existir o no.

El siguiente selector nos devolvería todos los usuarios que tengan el campo `nombre` definido.

```
{
    'nombre': { $exists: true }
}
```

Operadores lógicos de consulta

- `$and`: Une las cláusulas de búsqueda con un **and** lógico y devolverá los resultados que cumplan todas las cláusulas.
- `$or`: Une las cláusulas de búsqueda con un **or** lógico y devolverá los resultados que cumplan alguna de las cláusulas.
- `$nor`: Une las cláusulas de búsqueda con un **nor** lógico y devolverá los resultados que no cumplan ninguna de las cláusulas.
- `$not`: No es un operador como los anteriores. En el caso de `$not` **no se puede aplicar sobre un array de expresiones** si no que su función es la de negar una expresión.

No podrá usarse `$not` en expresiones como:

```
{
    $not: [ { 'rol': 'admin' }, { 'activo': true } ]
}
```

habrá que usar `$nor` en su lugar:

```
{
    $nor: [ { 'rol': 'admin' }, { 'activo': true } ]
}
```

la forma correcta de usar `$not` sería:

```
{
    'rol': { $not: { $eq: 'admin' } }
}
```

Ejemplo de `$and`:

```
{ $and: [ { <expression1> }, { <expression2> } , ... , { <expressionN> } ] }
```

Si quisiéramos seleccionar los usuarios que **no** sean administradores **y** que **no** estén activos lo haríamos de la siguiente manera:

```
{
    $nor: [ { 'rol': 'admin' }, { 'activo': true } ]
}
```

Operadores de comparación

Con estos operadores se pueden realizar consultas de comparación sobre los campos de los documentos. Toman como argumento un valor y devuelven los documentos que cumplan la condición.

- `$eq`: Igual a.
- `$ne`: Distinto de.
- `$gt`: Mayor que.
- `$gte`: Mayor o igual que.
- `$lt`: Menor que.
- `$lte`: Menor o igual que.
- `$in`: Igual a cualquiera de los valores de un array.
- `$nin`: Distinto a todos los valores de un array.

De este modo podríamos hacer una consulta para obtener los usuarios con una edad entre 18 y 65 años:

```
{
    'edad': { $gte: 18, $lte: 65 }
}
```

Consultas sobre *arrays*

Los campos cuyos valores son *arrays*, como veremos, tienen un comportamiento especial respectos a los campos *simples*.

Lo operadores específicos de consultas sobre *arrays* son:

- `$all`: Selecciona los documentos que contengan todos los valores de un array que especificamos.
- `$elemMatch`: Selecciona los documentos que contengan un elemento que cumpla las condiciones especificadas.
- `$size`: Selecciona los documentos que contengan un array con un número de elementos igual al especificado.

Adicionalmente **podremos hacer referencia a una posición concreta de un array** usando la notación de punto `<nombre campo array>.<posición>`:

```
db.alumnos.find( {
    'notas.0': { $gte: 5 },
} )
```

en el ejemplo anterior seleccionaríamos los alumnos cuya primera nota es mayor o igual que 5.

Seleccionar por igualdad en array

```
db.alumnos.find( { modulos: ['BDA', 'SBD'] } )
```

En esta consulta hay que tener en cuenta que **importa el orden** de los elementos del array. **Estamos comparando mediante una igualdad**.

Seleccionar por contenido del array

Si lo que queremos es seleccionar los documentos cuyo array `modulos` contiene un elemento lo haremos de la siguiente manera:

```
db.alumnos.find( {
    modulos: 'SBD'
} )
```

sería equivalente al siguiente código usando `$elemMatch`:

```
db.alumnos.find( {
    modulos: { $elemMatch: { $eq: 'SBD' } }
} )
```

Nótese que no se compara con ningún array, si no que se comprueba que el array incluye un elemento con la cadena `SBD`.

Si queremos comprobar si contiene varios elementos a la vez (y posiblemente otros) usaremos el operador `$all`.
El operador `$all` permite seleccionar los documentos que contengan **todos los valores** de un array que especificamos. Lo único que se comprueba es que el array contenga todos los elementos indicados sin importar el orden y sin importar que contenga otros elementos.

De esta forma con el siguiente filtro se devolverán los documentos que contengan los valores `matemáticas` y `física` en su array `especialidades`.

```
db.profesores.find( {
    especialidades: {
        $all: ['matemáticas', 'física']
    }
} )
```

O los alumnos que cursan los módulos `BDA` y `SBD`:

```
db.alumnos.find( {
    modulos: {
        $all: ['BDA', 'SBD']
    }
} )
```

Filtros compuestos

Con estos filtros lo que hacemos es poner condiciones a **algún** elemento del array. Así el siguiente código:

```
const cursor = db.inventory.find( {
  dim_cm: { $gt: 15, $lt: 20 }
} )
```

Seleccionará los documentos cuyo array `dim_cm` contenga **algún elemento** cuyo valor sea estrictamente mayor que 15 y menor que 20. Es decir, sería equivalente a:

```
const cursor = db.inventory.find( {
  dim_cm: {
    $elemMatch: {
      $gt: 15,
      $lt: 20
    }
  }
} )
```

Filtro para TODOS los elementos de un array

Supongamos que tenemos la siguiente colección en nuestra base de datos:

```
[
    {
        'id': 1,
        'notas': [5, 6, 8]
    },
    {
        'id': 2,
        'notas': [4, 5, 7]
    }
]
```

Si queremos seleccionar todos los documentos cuyas notas, **todas**, sean mayores o iguales a 5 hemos de escribir el siguiente código:

```
db.alumnos.find( {
    notas: { $not: { $lt: 5 } }
} )
```

Es decir, hemos de seleccionar los documentos cuyo array `notas` **no** contenga ningún elemento que sea menor que 5.

Puesto que no hay un operador que exija que todos los elementos cumplan una condición, lo que habrá que hacer es exigir que no haya algún elemento que cumpla la condición opuesta.

Pongamos otro ejemplo, si queremos ver qué alumnos nunca han alcanzado las notes de de 9 y 10:

```
db.alumnos.find( {
   "notas": {
      $not: {
         $all: [5, 6]
      }
   }
} )
```

Consultas sobre documentos embebidos

Lo único que hemos de tener en cuenta para realizar consultas sobre documentos embebidos se usa la notación de punto.

Si tenemos un documento con la siguiente estructura:

```
{
   'id': 1,
   'nombre': 'Manuel',
   'apellidos': 'Piñeiro',
   'dirección': {
      'calle': 'Calle de la Rosa',
      'número': 1,
      'piso': 2,
      'puerta': 'A'
   }
}
```

para hacer una consulta sobre el campo `calle` del documento embebido `dirección` escribiríamos:

```
db.alumnos.find( {
    'dirección.calle': 'Calle de la Rosa'
} )
```

Update / Actualizar

Para modificar el contenido de uno o más documentos de una colección usaremos los métodos `updateOne` o `updateMany`

`db.<nombre de la colección>.updateOne(<json selector>, <json de actualización>)`

Atomicidad

En MongoDB una operación de escritura es atómica a nivel de un único documento. Aún cuando la operación modifique múltiples documentos embebidos en dicho documento.

Cuando una única operación de escritura modifique múltiples documentos la modificación de **cada documento** es atómica pero la operación **en su conjunto** no lo es.

Operadores a nivel de campos

Estos operadores, como el título indica, realizarán cambios en los campos de un documento. Se clasifican de esta manera para diferenciarlos de los operadores que sirven para modificar *arrays*.

Operador `$set`

El operador `$set` reemplaza el valor de un campo por el valor especificado. Si el campo no existe lo crea.

Sintaxis de `$set`

```
{ $set: { <campo1>: <valor1>, ... } }
```

Si queremos especificar un campo dentro de un documento embebido hemos de usar la notación con punto.

Ejemplo de `$set`

```
db.alumnos.updateMany({}, {
    $set: {
        activo: false,
        modulos: []
    }
})
```

Operador `$unset`

Este operador se utilizará para eliminar campos de un documento.

Sintaxis de `$unset`

```
{ $unset: { <campo1>: <valor>, <campo2>: <valor>} }
```

Los valores de los campos se ignoran.

Ejemplo de `$unset`

El siguiente comando cambiará el estado de los alumnos cuyo curso sea igual a `2022/2023` a `inactivo` y eliminará el campo `notas`.

```
db.alumnos.updateMany( { curso: '2022/2023'}, {
    $set: { estado: 'inactivo' },
    $unset: { notas: 0 }
} )
```

Operadores `$inc` y `$mul`

El operador `$inc` *incrementa* el valor de un campo un determinado valor.
Este operador admite valores tanto positivos como negativos.

El operador `$mul`, por su parte, multiplica el valor de un campo por un determinador valor.

Sintaxis de `$inc`

```
{
    $inc: { <campo1>: <cantidad a incrementar>, <campo2>: <cantidad a
incrementar>, ... }
    $mul: { <campo3>: <multiplicador>, <campo4>: <multiplicador>, ... }
}
```

Si el campo a modificar no existe se crea. En ambos casos se crea con el valor inicial 0 y luego se aplica la modificación.

Operadores `$max` y `$min`

Estos operadores **sólo** modificarán el valor de un campo **si el nuevo valor es mayor**, en el caso de `$max`, **o menor**, en el caso de `$min`.

Sintaxis de `$max`

```
{
    $max: { <campo1>: <nuevo valor>, <campo2>: <nuevo valor> },
    $min: { <campo3>: <nuevo valor>, <campo4>: <nuevo valor> }
}
```

Si el campo no existe se crea y se le asigna el valor indicado.

Operador `$rename`

Sirve para renombrar campos.

Sintaxis de `$rename`

```
{
    $rename: { <campo1>: <nuevo nombre>, ... }
}
```

Operador `$currentDate`

Este operador es algo más complejo que los anteriores. Sirve para modificar un campo al valor de la fecha actual.

Sintaxis de `$currentDate`

```
{
    $currentDate: {
        <campo1>: true | { $type: 'date' } | { $type: 'timestamp' }
    }
}
```

Si el valor del campo es `true` se actualiza al valor de `Date` del momento actual. Si indicamos `$type` `timpestamp` se creará una marca de tiempo `Timestamp`. En el caso de que el valor se `date` se creará `Date`.

Si los campos no existen se crean.

Operador `$setOnInsert`

Este operador sólo tiene sentido si indicamos la opción `upsert: true`. Esta opción indica que si el selector no selecciona ningún documento se cree uno nuevo con los valores que se indique en la modificación.

Mediante el operador `$setOnInsert` indicaremos valores *por defecto* para campos que **sólo se crearán** si estamos insertando un nuevo documento como resultado del `upsert: true`.

Sintaxis de `$setOnInsert`

```
{
    $setOnInsert: { <campo1>: <valor1>, ... }
}
```

Operadores para *arrays*

En este apartado veremos los operadores para actuar sobre *arrays* y también sus modificadores (en realidad los modificadores son sólo para `$push`).

- `$`: Actúa como un *placeholder* para actualizar **el primer elemento** del array que cumpla las condiciones del selector.
- `$[]`: Actúa como un *placeholder* para actualizar **todos los elementos** que cumplan la condición del selector.

- `$[<identificador>]`: Actúa como un *placeholder* para actualizar **todos los elementos** que cumplan la condición de los ***arrayFilters*** para los documentos que hayan sido seleccionados por el selector.
- `$addToSet`: Añade valores a un array si no se encuentran ya en el mismo.
- `$pop`: Elimina el primer elemento del array.
- `$pull`: Elimina **todos los elementos** del array que cumplan la condición del selector.
- `$push`: Añade elementos al array.
- `$pullAll`: Elimina todos los elementos de un array.

Modificadores de los operadores de actualización

- `$each`: Se aplica al operador `$push` y `$addToSet` para añadir, uno a uno, varios elementos de un array.
- `$position`: Se aplica a `$push` para indicar la posición del array donde se añadirán los elementos.
- `$slice`: Se aplica a `$push` para limitar el tamaño del array modificado.
- `$sor`: También se aplica a `$push` para reordenar los documentos almacenados en un array.

Operador `$`

Este operador sirve para **hacer referencia al primer elemento** de un array que cumpla la condición expresada en el selector. Se utiliza de la siguiente manera:

```
db.alumnos.updateOne(
    { notas: { $lt: 5 } },
    { $set: { 'notas.$': 5} }
)
```

En el código anterior cambiaremos el valor del primer elemento del array cuyo valor sea menor que 5 a 5.

Es **obligatorio** que, si queremos modificar el array `notas` éste debe de aparecer en el selector.

Operador `$[]`

Este operador afectará a **todos los elementos** del array del documento que cumplan con las condiciones del selector.

Si modificamos el ejemplo anterior:

```
db.alumnos.updateOne(
    { notas: { $lt: 5 } },
    { $set: { 'notas.$[]': 5} }
)
```

Ahora, en lugar de modificar el primer elemento del array que cumpla la condición, modificará **todos los elementos** cuyo valor sea menor que 5.

Operador `$[<indentificador>]`

En este caso hemos de usar un **documento de opciones** con la opción `arrayFilters`. Esta opción sirve para aplicar *filtros* etiquetados (de ahí el `identificador`) para modificar sólo los elementos que pasen el filtro.

Ejemplo del operador `$[<indentificador>]`

El siguiente código:

```
db.alumnos.updateMany(
    { },
    { $set: { 'notas.$[cond1]': 10} },
    { arrayFilters: [ { cond1: { $eq: 5 } } ] }
)
```

1. Aplicará el selector `{ }` de manera que seleccionará todos los documentos de la colección.
2. A continuación empezará a ejecutar la operación `$set` del *update document*.
3. En dicha operación se hace referencia al filtro `cond1` que implica la igualdad de un elemento del array con el valor 5.
4. Se aplicará la operación `$set` a los elementos del array `notas` que cumplan que su valor es 5.

Operador `$push`

El operador `$push` añade uno o más valores a un array.

Sintaxis de `$push`

```
{ $push: { <campo1>: <valor1>, ... } }
```

El siguiente código:

```
db.alumnos.updateOne(
    { _id: 1 },
    { $push: { notas: 10 } }
)
```

Añadirá el valor 10 al array `notas` del documento cuyo `_id` sea 1.

Nótese que se añade **un elemento**. El código:

```
db.alumnos.updateOne(
    { _id: 1 },
    { $push: { notas: [10, 9] } }
)
```

Añadirá un elemento al array `notas` del documento cuyo `_id` sea 1. Este elemento será un array con los valores 10 y 9 y el documento resultante será:

```
{
    "_id" : 1,
    "nombre" : "Manuel",
    "apellidos" : "Piñeiro",
    "notas" : [ 5, 6, 8, [ 10, 9 ] ]
}
```

Si lo que queremos es añadir todos los elementos de un array a otro array hemos de utilizar el modificador `$each`.

```
db.alumnos.updateOne(
    { _id: 1 },
    { $push: { notas: { $each: [10, 9] } } }
)
```

que dará como resultado:

```
{
    "_id" : 1,
    "nombre" : "Manuel",
    "apellidos" : "Piñeiro",
    "notas" : [ 5, 6, 8, 10, 9 ]
}
```

Para indicar un campo en un **documento embebido** se puede usar la **notación de punto**.

Modificadores

- `$each`: Añadirá múltiples valores al array.
- `$slice`: Limita el número de elementos a añadir. Debe usarse con el modificador `$each`.
- `$sort`: Ordena los elementos del array. Se usará junto al modificador `$each`. Opciones:
    - Si insertamos elementos simples el valor `1` indicará orden ascendente y `-1` descendente.
    - Si insertamos documentos habrá que indicar el campo y luego especificar `1` o `-1` para indicar orden ascendente o descendente.
- `$position`: Especifica la posición del array en la que insertar los nuevos elementos. Requiere el uso de `$each`. Si no se usa `$position` `$push` insertará el elemento al final del array.

Ejemplo de sort:

```
{ $push: {
  <campo1>: {
      $each: [ { <c1>: 1, <c2>: 'a'}, { <c1>: 2, <c2>: 'b'}],
      sort: { <c1>: -1}
  }
}}
```

Sintaxis de los modificadores

```
{ $push: { <campo1>: { <modificador1>: <valor1>, ...}, ... } }
```

La forma de procesarse los modificadores será la siguiente:

1. Actualizar el array para añadir los elementos en la posición correcta.
2. Aplicar la ordenación si se especificó.
3. *Recortar* el array si se especificó.
4. Guardar el array.

Ejemplos de uso

Creamos la colección `estudiantes`:

```
db.estudiantes.insertOne( { _id: 1, notas: [ 44, 78, 38, 80] } )
```

Añadir un valor al array

El siguiente código añadirá el valor 89 al array `notas`:

```
db.estudiantes.updateOne( { _id: 1 }, { $push: { notas: 89 } } )
```

Añadir un valor a un array en múltiples documentos

Añadiremos el valor 89 al array `notas` de todos los estudiantes.

```
db.estudiantes.updateMany( {}, { $push: { notas: 89 } } )
```

Añadir múltiples valores a un array

```
db.estudiantes.updateOne( { _id: 1 }, { $push: {notas: {$each: [ 90, 92, 95 ] } }
} )
```

Utilizar `$push` con múltiples modificadores

Crearemos un nuevo estudiante para el ejemplo:

```
db.estudiantes.insertOne(
    {
    "_id" : 5,
        "quizzes" : [
            { "wk": 1, "score" : 10 },
            { "wk": 2, "score" : 8 },
            { "wk": 3, "score" : 5 },
            { "wk": 4, "score" : 6 }
        ]
    }
)
```

Haremos lo siguiente:

- Usaremos el operador `$each` para añadir cada `quiz` del array al array `quizzes` del documento.
- Usaremos `$sort` para ordenar dicho array `quizzes` de manera **descendiente** (argumento -1).
- Usamos `$slice` para quedarnos únicamente con los tres primeros elementos de `quizzes`.

```
db.estudiantes.updateOne(
    { _id: 5},
    {
        $push: {
            quizzes: {
                $each: [ { wk: 5, score: 8 }, { wk: 6, score: 7 }, { wk: 7,
score: 6 } ],
                $sort: { score: -1 },
                $slice: 3
            }
        }
    }
)
```

Ejemplo: Si tenemos un documento de la colección "usuarios" con el siguiente formato:

```
{
    id: 1,
    nombre: "Manuel",
    apellidos: "Piñeiro",
    fecha_nacimiento: "1977-05-07"
}
```

Para actualizar `apellidos` y `fecha_nacimiento` a "Piñeiro Mourazos" y "1977-05-15" respectivamente, usaríamos el siguiente comando:

```
db.usuarios.updateOne({ id: 1 }, { apellidos: "Piñeiro Mourazos",
fecha_nacimiento: "1977-05-07" })
```

Nota: se puede añadir atributos.

Delete / Eliminar

Esta operación se realiza con los métodos `deleteOne` o `deleteMany` y los filtros / selectores de MongoDB.

```
db.<nombre de la colección>.deleteOne(<documento de consulta>, <documento de
opciones>)
```

Veamos un par de ejemplos:

```
db.airbnb_bar.deleteOne( { _id: 1 } )
```

o

```
db.airbnb_bar.deleteMany(
    {
        host_neighbourhood: "Dreta de l'Eixample"
    },
    {
        writeConcern: {
            w: 1,
            j: true
        }
    }
 )
```

Estos comandos tienen como **valor de retorno** un documento (JSON) con dos campos:

- ***acknowledged***: un valor booleano que será cierto si la operación se ejecutó con *write concern* o falso si no lo hizo.
- ***deleteCount***: Indica el número total de documentos borrados.