Creando collections.

Hasta el momento hemos visto que las collection son creadas "on-demand", es decir, que se crean automáticamente a la hora de realizar el primer insert.
Sin embargo no es la única manera de crear una collection, ya que también podemos hacer uso del método "createCollection" del objeto global "db".

```
db.createCollection(<name>, { capped: <boolean>,
                              autoIndexId: <boolean>,
                              size: <number>, (capped)
                              max: <number>, (capped)
                              storageEngine: <document>,
                              validator: <document>,
                              validationLevel: <string>,
                              validationAction: <string>,
                              indexOptionDefaults: <document> } )
```

El primer argumento es el nombre de la collection a crear, el segundo es un objeto con opciones.

Option	Description.
capped	Establece que la collection será del tipo capped.
autoIndexId	Boolean, indica si el index _id se creará automáticamente o no. Por deprecar.
size	En las collection con capped: true, size indica en bytes la longitud máxima de la colleciton.
validator	Espera como argumento un objeto con las reglas para validar los documentos a almacenar dentro de la collection.
validationLevel	String, indica el nivel de validación a aplicar sobre los documentos. ● "off": indica que no se aplican validaciones. ● "strict": Default, indica que se deben validar todos los documentos en todas las operaciones de

	insert/update. ● "moderate": aplicar las reglas de validación nuevos documentos y para el update de documentos existentes, pero no sobre los existentes no válidos.

Document validator.

Cuando creamos una nueva collection tenemos la posibilidad de indicar un "document validator" con el fin de forzar a que cada nuevo documento a insertar dentro de la collection debe ser "validado" según un esquema que indiquemos previamente.
Para poder indicar el esquema contra el que validaremos cada nuevo documento, debemos utilizar la opción "validator" al momento de crear la collection.

```
db.createCollection( "users",
   { validator: { $and:
      [
         { phone: { $type: "string" } },
         { email: { $regex: /@gmail\.com$/ } },
         { status: { $in: [ "Complete", "Incomplete" ] } }
      ]
   }
});
```

En ejemplo anterior creamos la collection "users" con un esquema para validar cada uno de los futuros documentos.
Las reglas de nuestro validador son:

- Que el documento contenta la propiedad "phone" con un valor del tipo "string".
- Que el documento contenga la propiedad "email", el cual deberá terminar en "@gmail.com".
- Que el documento contenga la propiedad "status", cuyos posibles valores son "Complete" o "Incomplete".

Ahora vamos algunas pruebas de insert.

```
> db.contacts.insert({phone: "123123", email:"some@mongodb.com", status: "Complete"})
WriteResult({
        "nInserted" : 0,
        "writeError" : {
                "code" : 121,
                "errmsg" : "Document failed validation"
        }
```

```
})
```

En el ejemplo anterior MongoDB nos indica que no se ha podido insertar el nuevo documento debido a que no pudo pasar la validación (el email no termina con "@gmail.com").

```
> db.contacts.insert({phone: "123123", email:"some@mongodb.com", status: "Incomplete"})
WriteResult({ "nInserted" : 1 })
```

En el ejemplo anterior se pudo insertar sin problemas el nuevo documento.

Capped collections.

Las collecitons del tipo capped son aquellas que tendrá una longitud máxima establecida (ya sea en tamaño a nivel bytes o en máximos de documentos), y que una vez alcanzado ese máximo, se eliminarán automáticamente los documentos más viejos para poder guardar los documentos más nuevos.
Básicamente opera como una especie de buffer circular, en el cual se elimina el primer documento insertado para almacenar el nuevo documento.

Creando capped collection.

Para que la collection sea del tipo "capped", es necesario indicarlo en el momento de su creación.

```
db.createCollection("alumnos", {capped: true, size: 500000, max: 5});
var t = db.alumnos;
```

En el ejemplo anterior hemos creado la collection "alumnos" del tipo capped, con una longitud máxima a nivel bytes de 500000 o 5 documentos.

Cada vez que indicamos que una collection es del tipo capped, debemos indicar su longitud máxima en bytes. El poder indicar al máximo de documentos gracias a la propiedad "max" es opcional.

A continuación insertamos 5 documentos dentro de la collection "alumnos" y veamos como luce la collection al final.

```
> t.insert({added: new Date(), name: 'Alejandro', age:23});
WriteResult({ "nInserted" : 1 })
> t.insert({added: new Date(), name: 'Martin', age:22});
WriteResult({ "nInserted" : 1 })
> t.insert({added: new Date(), name: 'Carlos', age:18});
```

```
WriteResult({ "nInserted" : 1 })
> t.insert({added: new Date(), name: 'Juan', age:31});
WriteResult({ "nInserted" : 1 })
> t.insert({added: new Date(), name: 'Pedro', age:28});
WriteResult({ "nInserted" : 1 })
> db.alumnos.find({}, {_id:0})
{ "added" : ISODate("2016-08-18T19:09:44.324Z"), "name" : "Alejandro", "age" : 23 }
{ "added" : ISODate("2016-08-18T19:09:44.328Z"), "name" : "Martin", "age" : 22 }
{ "added" : ISODate("2016-08-18T19:09:44.330Z"), "name" : "Carlos", "age" : 18 }
{ "added" : ISODate("2016-08-18T19:09:44.331Z"), "name" : "Juan", "age" : 31 }
{ "added" : ISODate("2016-08-18T19:09:44.333Z"), "name" : "Pedro", "age" : 28 }
>
```

Ahora insertemos un nuevo alumno y veamos cómo se ve la collection.

```
> t.insert({added: new Date(), name: "Santiago", age:21});
WriteResult({ "nInserted" : 1 })
> db.alumnos.find({}, {_id:0})
{ "added" : ISODate("2016-08-18T19:09:44.328Z"), "name" : "Martin", "age" : 22 }
{ "added" : ISODate("2016-08-18T19:09:44.330Z"), "name" : "Carlos", "age" : 18 }
{ "added" : ISODate("2016-08-18T19:09:44.331Z"), "name" : "Juan", "age" : 31 }
{ "added" : ISODate("2016-08-18T19:09:44.333Z"), "name" : "Pedro", "age" : 28 }
{ "added" : ISODate("2016-08-18T19:12:24.717Z"), "name" : "Santiago", "age" : 21 }
>
```

Como podemos observar, el primer documento con el alumno "Alejandro" ya no existe en la collection, pero si existe el último documento con el alumno "Santiago".

Cursor tailable.

Un curso "tailable" es aquel cursor que contiene en su interior los nuevos documentos insertados en la collection (la cual debe ser del tipo capped). Esto quiere decir que una vez que alcanzamos el final del cursor (no existe más documentos), pero insertamos un documento nuevo en la collection, éste será accesible desde el cursor.

En el siguiente ejemplo creamos una collection "alumnos" del tipo capped, le seteamos un límite de 5 documentos, crear un cursor, lo mostramos en la consola, luego insertamos un nuevo documento y finalmente volvemos a ver el cursor.

```
> db.alumnos.drop();
true
> db.createCollection("alumnos", {capped: true, size: 500000, max: 5});
{ "ok" : 1 }
> var t = db.alumnos;
> t.insert({added: new Date(), name: 'Alejandro', age:23});
```

```
WriteResult({ "nInserted" : 1 })
> t.insert({added: new Date(), name: 'Martin', age:22});
WriteResult({ "nInserted" : 1 })
> t.insert({added: new Date(), name: 'Carlos', age:18});
WriteResult({ "nInserted" : 1 })
> t.insert({added: new Date(), name: 'Juan', age:31});
WriteResult({ "nInserted" : 1 })
> t.insert({added: new Date(), name: 'Pedro', age:28});
WriteResult({ "nInserted" : 1 })
> var results = t.find().addOption(DBQuery.Option.tailable);
> results;
{ "_id" : ObjectId("57b615c472debe733d5c3183"), "added" : ISODate("2016-08-18T20:08:36.440Z"),
"name" : "Alejandro", "age" : 23 }
{ "_id" : ObjectId("57b615c472debe733d5c3184"), "added" : ISODate("2016-08-18T20:08:36.444Z"),
"name" : "Martin", "age" : 22 }
{ "_id" : ObjectId("57b615c472debe733d5c3185"), "added" : ISODate("2016-08-18T20:08:36.446Z"),
"name" : "Carlos", "age" : 18 }
{ "_id" : ObjectId("57b615c472debe733d5c3186"), "added" : ISODate("2016-08-18T20:08:36.447Z"),
"name" : "Juan", "age" : 31 }
{ "_id" : ObjectId("57b615c472debe733d5c3187"), "added" : ISODate("2016-08-18T20:08:36.449Z"),
"name" : "Pedro", "age" : 28 }
> print("Cursor empty");
Cursor empty
> t.insert({added: new Date(), name: "Santiago", age:21});
WriteResult({ "nInserted" : 1 })
> results;
{ "_id" : ObjectId("57b615c472debe733d5c3188"), "added" : ISODate("2016-08-18T20:08:36.458Z"),
"name" : "Santiago", "age" : 21 }
>
```

GeoSpatial Index.

Los índices geoespaciales son aquellos que nos permiten realizar operaciones sobre el plano, es decir, poder realizar cálculos de proximidad, contención, etc.
Antes de empezar a trabajar con los índices geoespaciales es necesario que definamos el tipo de superficie con el que deseamos operar.
MongoDB almacena objetos del tipo GeoJSON para poder realizar las operaciones geoespaciales.
En MongoDB disponemos de dos índices principales para las operaciones geoespaciales: "2dsphere" y "2d".

GeoJSON.

GeoJSON es un estándar para poder representar en un JSON estructura de datos geográficas. Nos permite representar distintos tipos de objetos, cada uno de ellos con una estructura distinta.

Tipos de estructuras.

Point

```
 { "type": "Point", "coordinates": [-32.251, 30.5726] }
```

LineString

```
{ "type": "LineString",
    "coordinates": [ [100.0, 0.0], [101.0, 1.0] ]
    }
```

Polygon

```
{ "type": "Polygon",
    "coordinates": [
      [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0] ]
      ]
   }
```

MultiPoint

```
{ "type": "MultiPoint",
    "coordinates": [ [100.0, 0.0], [101.0, 1.0] ]
    }
```

MultiLineString

```
{ "type": "MultiLineString",
    "coordinates": [
        [ [100.0, 0.0], [101.0, 1.0] ],
        [ [102.0, 2.0], [103.0, 3.0] ]
      ]
    }
```

MultiPolygon

```
{ "type": "MultiPolygon",
    "coordinates": [
      [[102.0, 2.0], [103.0, 2.0], [103.0, 3.0], [102.0, 3.0], [102.0, 2.0]]],
      [[100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0]],
       [[100.2, 0.2], [100.8, 0.2], [100.8, 0.8], [100.2, 0.8], [100.2, 0.2]]]
      ]
    }
```

GeometryCollection

```
{ "type": "GeometryCollection",
    "geometries": [
      { "type": "Point",
        "coordinates": [100.0, 0.0]
        },
      { "type": "LineString",
        "coordinates": [ [101.0, 0.0], [102.0, 1.0] ]
        }
    ]
  }
```

2dsphere.

Cuando definimos que nuestra superficie será esférica, la misma se comportará como si se tratara del planeta tierra.
Para poder trabajar con una superficie esférica es necesario crear un índice del tipo "2dsphere" sobre una propiedad en la cual almacenaremos el registro geoJSON.

Veamos un ejemplo:

```
db.places.insert(
   {
      loc : { type: "Point", coordinates: [ -73.97, 40.77 ] },
      name: "Central Park",
      category : "Parks"
   }
)

db.places.insert(
   {
      loc : { type: "Point", coordinates: [ -73.88, 40.78 ] },
      name: "La Guardia Airport",
      category : "Airport"
   }
```

```
)

db.places.createIndex( { loc : "2dsphere" } )
```

Querys.

Existen distintos operadores para poder realizar consultas sobre los índices geoespaciales. Los operadores operan sobre los dos tipos de índices principales (Spherical y Flat).

Query Type	Geometry Type	Note
`$near (GeoJSON point, 2dsphere index)`	Spherical	
`$near (legacy coordinates, 2d index)`	Flat	
`$nearSphere (GeoJSON point, 2dsphere index)`	Spherical	
`$nearSphere (legacy coordinates, 2d index)`	Spherical	Use GeoJSON points instead.
`$geoWithin : { $geometry: ... }`	Spherical	
`$geoWithin : { $box: ... }`	Flat	
`$geoWithin : { $polygon: ... }`	Flat	
`$geoWithin : { $center: ... }`	Flat	
`$geoWithin : { $centerSphere: ... }`	Spherical	
`$geoIntersects`	Spherical	

$near.

```
{
  $near: {
     $geometry: {
        type: "Point" ,
```

```
        coordinates: [ <longitude> , <latitude> ]
     },
     $maxDistance: <distance in meters>,
     $minDistance: <distance in meters>
  }
}
```

Pokemon Example.

En el siguiente ejemplo veremos cómo crear un simulador del famoso juego Pokemon Go

Create the index.

```
db.markers.createIndex( { location : "2dsphere" } );
```

$near.

```
db.markers.find(
    {
        location: {
            $near: {
                $geometry: {
                    type: "Point" ,
                    coordinates: [ -58.461531, -34.569751]
                },
                $maxDistance: 200
            }
        }
    }
)
```