UAX FP

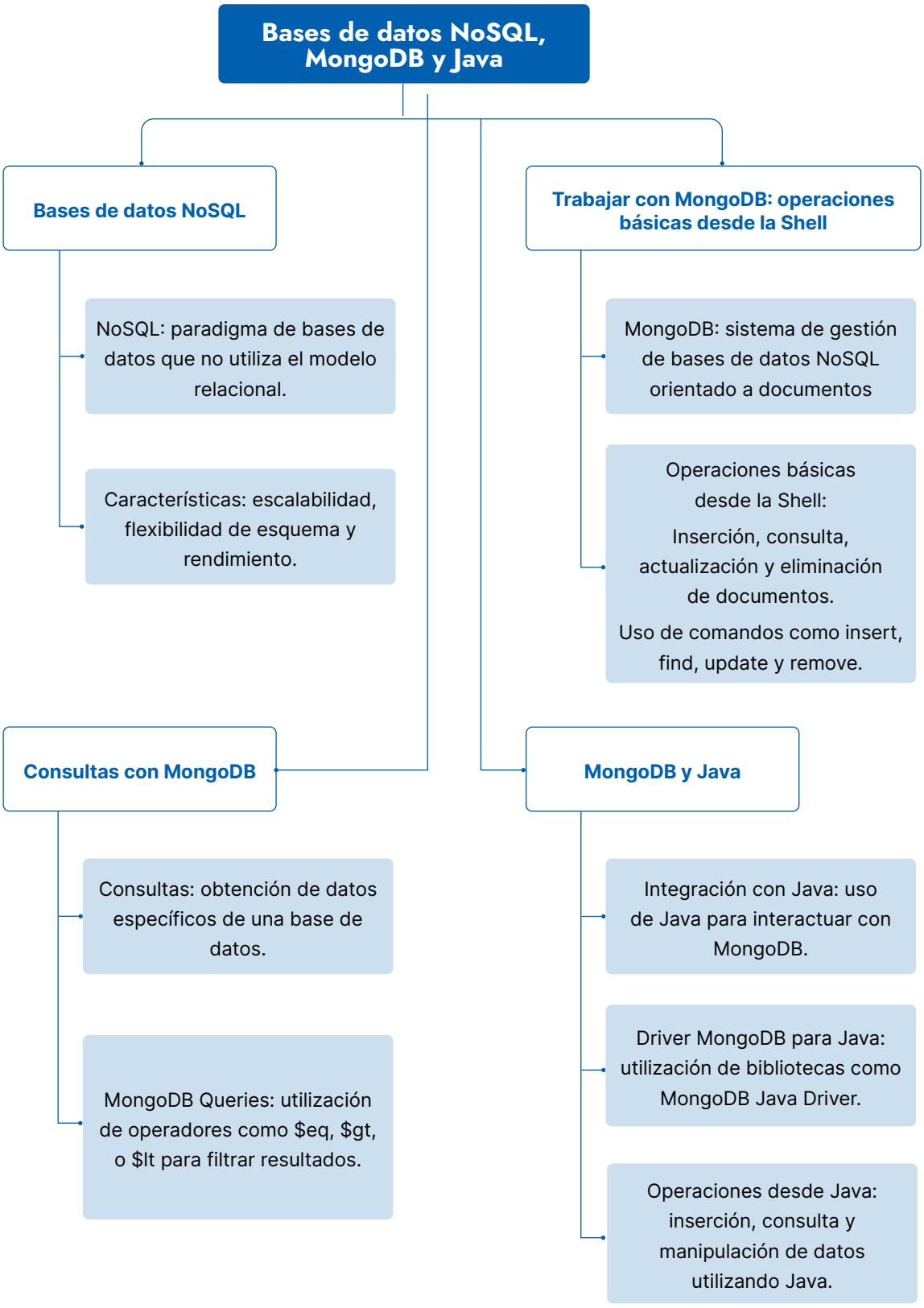
UF6

Bases de datos NoSQL, MongoDB y Java

Acceso a datos

ÍNDICE

Mapa conceptual



01 Bases de datos NoSQL

1.1. El movimiento NoSQL

Aunque hoy en día la mayoría de sistemas de gestión de bases de datos siguen basándose en el modelo relacional, cada vez van ganando más relevancia sistemas basados en otros paradigmas de base de datos, como las orientadas a objetos, o las basadas en el lenguaje **XML,** cuyas principales características han sido absorbidas en gran parte por los SGBD relacionales.

DEFINICIÓN

Bajo el término de **NoSQL** se engloban todas aquellas alternativas a los sistemas tradicionales que no utilizan el modelo relacional como base, aunque también se le ha dado el significado de ***not only SQL,*** en referencia a que algunos de estos sistemas sí que utilizan SQL como lenguaje de consulta, pese a no basarse en un modelo relacional.

¿SABÍAS QUE...?

Con la llegada de **la web 2.0**, el crecimiento de datos en Internet creció de forma exponencial, de la mano, principalmente, de las redes sociales y del contenido multimedia: Facebook, Twitter, YouTube, etcétera, plataformas donde son los propios usuarios quienes aportan la mayor parte de contenido, lo que provoca un rápido crecimiento de los datos almacenados y un serio problema para los sistemas que no estaban preparados para ello. Con la web 3.0, 4.0 y el auge de la inteligencia artificial, Internet se ha convertido en una gran base de datos, ya no solo para proporcionar contenido a la web, sino a cualquier tipo de aplicación.

Las **bases de datos NoSQL** permiten asumir estos escenarios donde las bases de datos resultan problemáticas, debido principalmente a la falta de escalabilidad (posibilidad de crecimiento) y bajo rendimiento de estas, cuando miles de usuarios acceden de forma simultánea a la información.

Todo esto no significa que las bases de datos tradicionales se vayan a reemplazar por las NoSQL, sino que estamos en un panorama donde no podemos encontrar una única tecnología que sea apropiada para todos los escenarios. Es por ello por lo que estamos hoy en día ante un buen número de soluciones específicas que mejoran ciertos problemas; todas diferentes, pero englobadas dentro del movimiento NoSQL.

1.2. Tipos de bases de datos NoSQL

Dentro del panorama NoSQL podemos encontrar diferentes **tipos de bases de datos, según su forma de almacenar los datos.** Entre ellas, podemos destacar las siguientes.

A. Bases de datos clave-valor

Se trata de un modelo de base de datos bastante sencillo y popular, donde cada elemento se identifica por una clave única, siguiendo el modelo de las tablas ***hash,*** de modo que el dato se recupera de forma muy rápida. Generalmente, los objetos se almacenan como objetos binarios (BLOB).
Algunas bases de datos de este tipo son **Cassandra (Apache), Bigtable (Google)** o **Dynamo (Amazon).**

EJEMPLO

Supongamos que estamos construyendo una aplicación de comercio electrónico y queremos almacenar la información de los productos en una base de datos clave-valor. Podríamos tener una clave única para cada producto, como el ID del producto, y el valor sería el objeto binario que contiene toda la información del producto, como nombre, descripción, precio, etc.
Por ejemplo:
Clave: “123456”
Valor: Objeto binario que contiene información del producto con ID “123456”

B. Bases de datos documentales

Este modelo **almacena la información en forma de documentos, generalmente XML o JSON**, y se utiliza una clave única para cada registro, por lo que se permiten búsquedas por clave-valor. La diferencia respecto a las bases de datos clave-valor anteriores es que aquí el valor es el propio documento, no un dato binario.
Como veremos más adelante, son muy versátiles, de modo que no necesitamos tan siguiera tener una estructura común a los documentos que guardamos.
El máximo exponente de este tipo de bases de datos es **MongoDB.**

EJEMPLO

Por ejemplo:
Clave: “post_789” Valor:

```
json
{
   “title”: “Cómo construir una aplicación web con MongoDB”,
   “author”: “John Doe”,
    “content”: “En este tutorial aprenderás a construir una aplicación web utilizando MongoDB como base de datos...”,
   “tags”: [“MongoDB”, “Web Development”],
   “date”: “2024-05-07”
}
```

C. Bases de daetos en grafo

Un **grafo** es un conjunto de vértices o nodos unidos por aristas, que nos permiten representar relaciones entre ellos.
Las bases de datos en grafo pretenden seguir este modelo, de manera que la información se representa como nodos en un grafo, y las relaciones entre ellos se representan mediante aristas. De este modo, aprovechando la teoría de grafos, podemos recorrer la información de forma óptima.
Algunos ejemplos de este tipo de bases de datos son **Amazon Neptune**, **JanusGraph (Apache)**, **SQL Server (Microsoft)** o **Neo4j.**

EJEMPLO

Supongamos que estamos desarrollando una red social y queremos almacenar las relaciones entre usuarios en una base de datos en grafo. Cada usuario sería un nodo en el grafo y las relaciones entre ellos, como amistades o seguidores, serían representadas como aristas entre los nodos.
Por ejemplo:
Nodos:

- Usuario A
- Usuario B
- Usuario C

Relaciones:

- Usuario A es amigo de Usuario B
- Usuario B sigue a Usuario C

En este caso, cada nodo representaría un usuario y las aristas representarían las relaciones entre ellos, como amistades o seguidores. Esto permite un acceso eficiente a la información sobre las relaciones entre los usuarios.

1.3. Bases de datos documentales con MongoDB

Vamos a centrarnos en el estudio de uno de los sistemas de bases de datos NoSQL más utilizados hoy en día: MongoDB.

DEFINICIÓN

MongoDB es una base de datos orientada a documentos, basada en el almacenamiento de sus estructuras de datos en documentos de tipo JSON con un esquema dinámico. Aunque empezó siendo desarrollado por la empresa **10gen,** hoy en día es un proyecto de código abierto, con una gran comunidad de usuarios.

Un servidor MongoDB puede contener varias bases de datos, y cada una de ellas compuesta por un conjunto de colecciones, que podríamos equiparar a las tablas de una BD relacional. Cada colección almacena un conjunto de documentos JSON, formado por atributos clave-valor, que vendrían a ser los registros de una base de datos relacional.
A grandes rasgos, podríamos establecer las siguientes equiparaciones:

MODELO RELACIONAL	MONGODB
BD relacional	BD orientada a documentos
Tabla	Colección
Registro/fila	Documento JSON
Atributos/columnas	Claves del documento JSON

EJEMPLO

Veamos, a modo de ejemplo, una colección **Películas** con dos documentos:

```
[{
     _id: 1,
    titulo: "La Amenaza Fantasma",
     anyo: 1999,
     director: {
         nombre: "George",
         apellidos:  "Lucas",
         anyo_nacimiento: 1944
     }
},
{
      _id: 2,
    titol: "El Ataque de los Clones",
     year: 2002,
     director: {
         nombre: "George",
         apellidos:  "Lucas",
         anyo_nacimiento: 1944
     }
}]
```

Como podemos ver, cada documento de tipo película posee sus propios atributos, y estos no tienen por qué coincidir entre dos documentos. Otra característica interesante es que, como podemos observar, disponemos de toda la información sobre el director dentro del propio documento.

CARACTERÍSTICAS DE MONGODB

Algunas de las **principales características de MongoDB** son:

- Es una base de datos **orientada a documentos:** los diferentes documentos u objetos de la base de datos se pueden mapear fácilmente a los objetos de las aplicaciones.
- Como en todo documento JSON, el valor asociado a una clave puede ser también un documento JSON, lo que aporta mucha **flexibilidad.** Esto sería el equivalente a las tablas embebidas en las bases de datos objeto-relacionales. Este hecho de poder tener documentos **embebidos** en otros facilita las consultas, ya que no se necesita hacer `Joins`.

- Los **esquemas** de los documentos son **dinámicos.** Esto significa que dos documentos no tienen por qué seguir el mismo esquema (un documento puede tener campos diferentes a otros documentos), con lo que el polimorfismo resulta más sencillo.
- Proporciona un **alto rendimiento,** ya que los documentos embebidos posibilitan las lecturas y escrituras más rápidas, los índices pueden incluir claves de documentos embebidos y vectores, y, además, se pueden realizar escrituras en ***streaming*** cuando no sea necesario confirmar el final de la escritura.
- Proporcionan **alta disponibilidad,** con servidores replicados con gestión automática de errores del sistema.
- Es fácilmente **escalable**, con ***sharding*** (fragmentación) automático, que distribuye los datos de una colección entre varias máquinas, y las lecturas se pueden distribuir por los servidores replicados.

Respecto de los SGBD, podemos destacar las siguientes diferencias:

- MongoDB ofrece una **mayor flexibilidad,** al no estar sujeto a la estricta definición de un esquema. Por ejemplo, cuando en un SGBDR necesitamos un campo nuevo para una fila concreta, hay que incorporar el campo en todas las filas, mientras que, en MongoDB, cuando necesitamos añadir un campo a un documento, solo se lo tenemos que añadir al propio documento, sin que los demás se vean afectados.
- **No se utiliza el lenguaje SQL** para hacer consultas, sino que se pasa como parámetro un JSON que describe lo que se quiere recuperar.
- MongoDB **no admite los JOIN,** sino que ofrece tipos de datos multidimensionales, como vectores u otros documentos dentro del propio documento. Por ejemplo, un documento puede tener un vector con todos los objetos relacionados de otra colección.
- No es necesario realizar transacciones cuando las colecciones sean en un mismo documento, pues siempre se realiza de forma atómica. En el caso que se use distintos documentos, **MongoDB** (a partir de la versión 4.2) sí ofrece herramientas para poder realizar transacciones.

1.4. El ecosistema de MongoDB

VÍDEO LAB

Para consultar el siguiente vídeo sobre MongoDB en acceso a datos. NoSQL escanea el código QR o pulsa aquí.

MongoDB abarca un gran abanico de posibilidades, desde servidores para bases de datos locales hasta bases de datos en la nube. En su **web**, podemos descubrir los diferentes productos y servicios que se ofrecen, entre los que encontramos:

- **Servidor de BD MongoDB,** con sus dos versiones, la Community, versión comunitaria y gratuita, y la Enterprise, su versión comercial orientada al mundo empresarial y con características adicionales que mejoran el rendimiento y el soporte. Además del servidor en sí, en la web se ofrecen también el servidor preparado para su uso en contenedores, mediante operadores Kubernetes. El servidor está disponible en varias plataformas: Linux, Solaris, MacOS X y Windows.
- **MongoDB Atlas,** la plataforma MongoDB en la nube (DBaaS o ***database as a service***), que permite su despliegue en servicios como AWS, Azure o Google Coud.
- **Realm,** un servicio de datos pensado para aplicaciones móviles y web, y que incluye, además de BD en la nube, varios servicios de ***backend*** completamente administrados.

02 Trabajar con MongoDB: operaciones básicas desde la *shell*

2.1. Colecciones y documentos

IMPORTANTE

La unidad de información con que trabaja MongoDB es el documento, que sería el equivalente a un registro en un modelo relacional. Se trata de documentos JSON, formados por pares **clave-valor,** y que representan la información de forma bastante intuitiva. Los servidores MongoDB, por su parte, almacenarán estos datos en formato BSON (Binary JSON), un formato binario de serialización.

Respecto a los **documentos** JSON para MongoDB, hay que tener en cuenta algunos aspectos:

Respecto a las claves

- No pueden ser nulas.
- Pueden estar formadas por cualquier carácter UTF-8, salvo los caracteres «.» o «**$**».
- Son ***case-sensitive***.
- Deben ser únicas dentro de un mismo documento.

Respecto a sus valores

- Pueden ser de cualquier tipo permitido.

Respecto al documento

- Debe poseer un campo **_id**, con un valor único, que actuará como identificador del documento. Si no especificamos esta clave, MongoDB la generará automáticamente, con un objeto de tipo `ObjectId`.

Esquema Correcto:

Colecciones y documentos.

Restricciones y Características:

CLAVES
No pueden ser nulas.
Pueden ser cualquier carácter UTF-8 excepto ".", "$".
Son case-sensitive.
Deben ser únicas dentro de un mismo documento.

Valores
»Pueden ser de cualquier tipo permitido.

Documento
»Debe tener un campo “_id” único, que actúa como identificador del documento. Si no se especifica, MongoDB lo generará automáticamente con un ObjectId.

Esquema Incorrecto:

Colecciones y Documentos

Errores
»**Clave Nula:** Se muestra un documento con una clave nula. »**Nombre de Colección con Símbolo “$”:** Se muestra una colección con un nombre que contiene el símbolo “$”. »**Documento sin Campo “_id”:** Se muestra un documento sin el campo “_id”, que es obligatorio en MongoDB. Esto viola la regla de que cada documento debe tener un campo “_id” único.

IMPORTANTE

Si los documentos son el equivalente a los registros, **las colecciones son el equivalente a las tablas**, con la diferencia de que las colecciones poseen un esquema dinámico, con lo que documentos de la misma colección pueden presentar claves o tipos de datos diferentes entre ellos.

Los nombres de las colecciones estarán sujetas a las siguientes **restricciones**:

- No pueden ser la cadena vacía (“”), ni el carácter nulo, ni contener el símbolo `$`.

Podemos utilizar el punto (.) en los nombres de colecciones para añadir prefijos a esta, pero no se pueden crear colecciones con el prefijo **system.**, ya que este se usa para colecciones internas del sistema. Por ejemplo, **db.system.prueba** no sería válido, pero **db.systema.prueba**, sí.

2.2. Operaciones básicas con MongoDB

Los estudiantes han tenido la oportunidad de explorar y comprender los conceptos fundamentales a través de herramientas como MongoDB Compass. Esta familiaridad con MongoDB Compass les ha brindado una interfaz gráfica intuitiva para interactuar con bases de datos MongoDB, explorar colecciones, realizar consultas y comprender la estructura de los datos.

Ahora, es el momento de llevar esa experiencia al siguiente nivel y sumergirse en el corazón mismo de MongoDB: la base de datos MongoDB en sí misma. MongoDB es mucho más que una colección de documentos; es un sistema de gestión de bases de datos altamente flexible y escalable que ofrece una amplia gama de características y funcionalidades para satisfacer las necesidades de desarrollo de aplicaciones modernas.

Al pasar de MongoDB Compass a la base de datos MongoDB, se abren las puerta a un mundo de posibilidades en el desarrollo de aplicaciones. Aprender a diseñar esquemas de datos eficientes, a optimizar consultas para un rendimiento óptimo, a implementar estrategias de escalabilidad horizontal y a utilizar las características avanzadas de MongoDB, como la replicación, la fragmentación y la indexación.

El conocimiento adquirido a través de MongoDB Compass proporciona una base sólida para explorar y aprovechar todas las capacidades de MongoDB en el desarrollo de aplicaciones del mundo real. Desde aplicaciones web hasta sistemas de gestión de contenidos, análisis de datos y más, MongoDB ofrece una solución robusta y flexible para una amplia variedad de casos de uso.

En la siguiente tabla, vamos a ver algunas de las **operaciones** que podemos realizar sobre MongoDB:

OPERACIÓN	SIGNIFICADO	EJEMPLO
`insertOne(documento)` `insertMany(documentos)`	Añade un documento a la colección.	`db.coleccion.insertOne({ a:1 })`
`find(criterio)`	Obtiene todos los documentos de una colección que coinciden con el patrón indicado.	`db.coleccion.find({a:1});`
`findOne(Criterio)`	Obtiene un elemento de la colección coincidente con el patrón.	`db.coleccion.findOne();`
`updateOne(Criterio, Operacion, [opciones]` `updateMany(Criterio, Operacion, [opciones])`	Actualiza un documento de la colección (o varios, en el caso de **updteMany**). Requiere dos parámetros: el criterio de busca del documento que se va a actualizar y la operación de actualización. Admite un tercer parámetro opcional para las opciones.	`db.coleccion.updateOne({a:1}, {$set: {a:2}})`

`deleteOne(Criterio)` `deleteMany(Criterio)`	Borra los documentos de una colección que cumplen el criterio.	`db.coleccion.deleteOne({a:1})`

En los siguientes apartados profundizaremos en las diferentes operaciones.

2.3. Tipos de datos

Los tipos de datos con que trabaja MongoDB son similares a los que podemos encontrar en JavaScript. Aunque dispones de una descripción más detallada en la documentación adicional, vamos a examinar algunos de estos tipos:

A. Tipos básicos

MongoDB admite los **tipos básicos** que se describen en la siguiente tabla:

TIPO	DESCRIPCIÓN

null	Representa tanto el valor nulo como un campo que no existe.
	»**Descripción:** Representa tanto el valor nulo como un campo que no existe.
	»**Ejemplo:**
	`json`
	`{ "name": "John", "age": null }`
	En este ejemplo, el campo "age" tiene un valor nulo, lo que indica que no se ha proporcionado la edad de la persona.

boolean	Permite los valores `true` y `false`.
	»**Descripción:** Permite los valores true y false.
	»**Ejemplo:**
	`json`
	`{ "is_active": true, "is_admin": false }`
	Aquí, los campos "is_active" y "is_admin" indican si un usuario está activo y si es un administrador, respectivamente.

number	Representan valores numéricos en coma flotante. Si queremos utilizar tipos enteros o enteros largos, hay que utilizar las clases propias: `NumberInt` (32 bits) o `NumberLong` (64 bits).
	»**Descripción:** Representan valores numéricos en coma flotante.
	»**Ejemplo:**
	`json`
	`{ "price": 29.99, "quantity": 10 }`
	En este caso, el campo "price" representa el precio de un producto como un valor decimal, mientras que "quantity" representa la cantidad de ese producto en stock como un número entero.

String

Representan cualquier cadena de texto UTF-8 válida.

»**Descripción:** Representa cualquier cadena de texto UTF-8 válida.

»**Ejemplo:**

```
json
{ "name": "Alice", "email": "alice@example.com" }
```

Aquí, los campos "name" y "email" contienen cadenas de texto que representan el nombre y el correo electrónico de una persona, respectivamente.

Date

Representa fechas, expresadas en milisegundos.

»**Descripción:** Representa fechas, expresadas en milisegundos.

»**Ejemplo:**

```
json
{ "created_at": ISODate("2024-05-07T08:00:00.000Z") }
```

Este ejemplo muestra el campo "created_at" que contiene la fecha y hora en que se creó un registro, representada como un objeto Date de MongoDB.

array

Listas de valores que se representan como vectores.

»**Descripción:** Listas de valores que se representan como vectores.

»**Ejemplo:**

```
json
{ "tags": ["mongodb", "database", "nosql"] }
```

En este ejemplo, el campo "tags" es un array que contiene las etiquetas asociadas a un artículo o producto.

Documentos incrustados

Los documentos pueden tener otros documentos incrustados en **ellos.**

»**Descripción:** Los documentos pueden tener otros documentos incrustados en ellos.

»**Ejemplo:**

```
json
{
  "author": {
    "name": "Jane Doe",
    "age": 35,
    "email": "jane@example.com"
  }
}
```

Aquí, el campo "author" es un documento incrustado que contiene información sobre el autor de un artículo o publicación.

ObjectId	Se trata del tipo por defecto para los campos **_id**, y está diseñado para poder generar de forma sencilla valores únicos de forma global.
	»**Descripción:** Se trata del tipo por defecto para los campos _id, y está diseñado para poder generar de forma sencilla valores únicos de forma global.
	»**Ejemplo:**
	`json`
	`{ "_id": ObjectId("61234d4b5c7c8d9e0f1a2b3c") }`
	En este ejemplo, el campo "_id" contiene un ObjectId generado automáticamente que actúa como identificador único para el documento.

Otro ejemplo tipos de datos:

```
{
  "_id": ObjectId("61234d4b5c7c8d9e0f1a2b3c"),
  "name": "John",
  "age": null,
  "is_active": true,
  "balance": NumberLong("10000000000"),
  "registration_date": new Date(),
  "tags": ["mongodb", "database", "nosql"],
  "preferences": {
    "notifications": true,
    "theme": "dark"
  },
  "last_login": ISODate("2024-05-07T08:00:00.000Z"),
  "address": {
    "street": "123 Main St",
    "city": "Anytown",
    "zip": "12345"
  },
  "director": {
    "name": "Gareth",
    "surname": "Edwards",
    "birth_year": 1975,
    "nationality": "British"
  }
}
```

»**id:** Utiliza el tipo de dato ObjectId como identificador único del documento.

»**name**: Utiliza el tipo de dato **String** para representar el nombre de una persona.

»**age**: Utiliza el tipo de dato **null** para indicar que la edad no está disponible.

»**is_active**: Utiliza el tipo de dato **boolean** para indicar si el usuario está activo.

- **balance**: Utiliza el tipo de dato **NumberLong** para representar un número largo.
- **registration_date**: Utiliza el tipo de dato **Date** para representar la fecha de registro.
- **tags**: Utiliza un array para representar las etiquetas asociadas al usuario.
- **preferences**: Utiliza un documento incrustado para representar las preferencias del usuario.
- **last_login**: Utiliza el tipo de dato **Date** para representar la última vez que el usuario inició sesión.
- **address**: Utiliza un documento incrustado para representar la dirección del usuario.
- **director**: Utiliza un documento incrustado para representar información sobre un director de cine.

Vamos a ver algunas peculiaridades de interés sobre algunos de estos tipos.

B. El tipo Date

Mongo utiliza el **tipo Date** de JavaScript. Cuando generamos un nuevo objeto de tipo **Date**, hay que utilizar el operador **New**, puesto que en caso contrario obtendríamos una representación de la fecha en forma de ***string.***

Por ejemplo, si definimos las variables **a** y **b** del siguiente modo:

```
test> let a=Date()
test> let b=new Date()
```

Podemos ver que los resultados son bastante diferentes:

```
test> a Sun May 08 2022 06:46:01 GMT+0200 (hora de verano de Europa
central) test> typeof(a) string

test> b ISODate("2022-05-
08T04:46:09.371Z")
test> typeof(b) object
```

C. Vectores

Los vectores pueden utilizarse tanto para representar colecciones ordenadas, como listas o colas, o bien colecciones no ordenadas, como los conjuntos. Al igual que en JavaScript, y a diferencia de otros lenguajes, como Java, cada elemento del vector puede tener un tipo de dato diferente, incluso otros objetos de tipo vector.
Veamos algunos ejemplos sobre vectores en JavaScript y por tanto en MongoDB:

- Creación de un vector:

```
test> let v={objetos: ["casa", 10, {texto: "hola"}, false] }
```

- Consulta del vector o sus componentes:

```
test> v
{ objetos: [ 'casa', 10, { texto: 'hola' }, false ] }
test> v.objetos
[ 'casa', 10, { texto: 'hola' }, false ]
test> v.objetos[1]
10
test> v.objetos[2]
{ texto: 'hola' }
```

- Modificación de valores:

```
test> v.objetos[3]=!v.objetos[3]
true
test> v
{ objetos: [ 'casa', 10, { texto: 'hola' }, true ] }
```

D. Documentos incrustados o embebidos

Un **par clave-valor** en un documento puede tener como valor otro documento. Esto se conoce como documentos incrustados (**embed**), y no sería más que utilizar un objeto JSON dentro de otro.
Por ejemplo:

```
> let peli={
        titulo: "Rogue One. A Star Wars Story.",
        anyo: 2016,
       director: {
           nombre: "Gareth",
           apellidos:  "Edwards",
           anyo_nacimiento: 1975,
```

```
            nacionalidad: "británica"
        }
    }
```

Como vemos, el propio documento lleva información sobre la película y sobre su director. En un modelo relacional, normalmente tendríamos dos tablas relacionadas entre ellas. En este caso, es posible que, si deseamos mantener información específica sobre los directores, acabemos teniendo información redundante.

E. Sobre los ObjectIds

La clase `ObjectId` utiliza 12 bytes, organizados de la siguiente forma:

0	1	2	3	4	5	6	7	8	9	10	11
	Timestamp			Timestamp			Timestamp		Timestamp		

- ***Timestamp* (bytes 0-3)**: el ***timestamp*** en segundos desde el 1 de enero de 1970.
- ***Machine* (bytes 4-6)**: identificador único de la máquina; generalmente, un ***hash*** de su ***hostname***.
- **PID (bytes 7-8):** identificador del proceso que genera el **ObjectID**, para garantizar la unicidad dentro de la misma máquina.
- **Incremento (bytes 9-11):** valor autoincremental, para garantizar la unicidad en el mismo segundo, máquina y proceso.

IMPORTANTE

Como vemos, se trata de un mecanismo más robusto que un campo autoincremental como en MySQL. Esto se corresponde a la naturaleza distribuida de MongoDB, de forma que se puedan generar los objetos en un entorno con múltiples hosts.

2.4. Añadir información a las colecciones

La manera natural de añadir elementos a la base de datos es mediante los diferentes métodos `insert`, disponibles en todas las colecciones.

A. *INSERTONE()*

Permite insertar un único documento en la colección. Por ejemplo, para insertar el documento **peli** definido anteriormente, podríamos hacer:

```
test> db.misPelis.insertOne(peli)
{
  acknowledged: true,
  insertedId: ObjectId("6277510ab54867b80b742ddf")
}
```

Como vemos, la respuesta obtenida es un documento JSON que contiene un valor booleano que indica si la operación se realizó con éxito, y un **ObjectID**, con el ID asignado automáticamente.
Algunas consideraciones:

- Si la colección a la que añadimos un documento no existe, esta se crea de forma automática.
- Respecto al campo **_id**, como vemos, este se generó automáticamente. No obstante, podemos indicar nosotros dicho identificador, sin que este sea además de tipo **ObjectId**; la única restricción es que esta sea única, para evitar duplicados.
- Hay que destacar que no hemos utilizado ningún esquema para la colección, ya que cada documento que insertemos puede tener un esquema diferente.

B. *INSERTMANY()*

Permite añadir varios documentos en una colección. Para ello, le proporcionaremos un vector de **documentos** por añadir a la **colección**.
Por ejemplo, si definimos tres documentos nuevos del siguiente modo:

```
test> let peli2={titulo: "Star Wars. A new Hope", anyo: 1977};
test> let peli3={titulo: "Empire Strikes Back", anyo: 1981};
test> let peli4={titulo: "Return of the Jedi", anyo: 1984};
```

Podemos insertarlos con:

```
test> db.misPelis.insertMany([peli2, peli3, peli4])
{
  acknowledged: true,
```

```
  insertedIds: {
'0': ObjectId("627759a5b54867b80b742de0"),
'1': ObjectId("627759a5b54867b80b742de1"),
'2': ObjectId("627759a5b54867b80b742de2")
 }
}
```

CLAVES Y CONSEJOS

Hay que tener en cuenta que, cuando se produce un error en la inserción de un elemento del vector (por ejemplo, si hemos indicado una clave duplicada), finaliza la ejecución del proceso, de modo que ni el documento que ha producido el error ni los siguientes en el vector se insertarán a la colección.

2.5. Eliminar información de MongoDB

Para eliminar documentos de una colección usaremos las órdenes **`deleteOne()`**, **`deleteMany()`** o **`findOneAndDelete()`**, proporcionándoles como parámetro un JSON, con la condición que queremos que cumpla el documento o documentos que se quieren eliminar.

- La orden **`deleteOne`** eliminará únicamente el primer elemento que coincida con el criterio, de modo que si lo que deseamos es eliminar un documento concreto, deberemos utilizar criterios que se correspondan con índices únicos, como, por ejemplo, el **`_id`**.
- La orden **`deleteMany`** eliminará todos los documentos que coincidan con el criterio.

Tanto **`deleteOne`** como **`deleteMany`** devuelven un documento con un booleano, indicando si se ha realizado la operación, así como el número de elementos eliminados (**`deletedCount`**). Por su parte, **`findOneAndDelete`** también elimina un documento, de acuerdo con criterios de selección y ordenación, pero devolviendo además el documento que se ha eliminado. Por ejemplo, creamos una colección con varios elementos:

```
db.pruebas.insertMany([{x:1}, {x:2}, {x:3}, {x:4}, {x:5}, {x:6},
{x:7}]);
```

Ahora podemos:

- Borrar un elemento:

```
test> db.pruebas.deleteOne({}) {
acknowledged: true, deletedCount: 1 }
```

Con lo que habrá eliminado el primer registro.

- Eliminar varios elementos que cumplan un criterio; por ejemplo, que el valor de ***x*** es mayor que **3**:

```
test> db.pruebas.deleteMany({x:{$gt:3}})
{ acknowledged: true, deletedCount: 4 }
```

Como vemos, se han eliminado cuatro documentos. El criterio `{$gt:3}` lo veremos posteriormente, pero sirve para seleccionar aquellos registros cuyo valor en ***x*** sea mayor que **3**.

- Eliminar un documento y devolverlo:

```
test> db.pruebas.findOneAndDelete({x:2})
{ _id: ObjectId("6277687fb54867b80b742deb"), x: 2 }
```

Por otro lado, también podemos eliminar una colección completa mediante la orden **`drop`**. Hay que tener especial cuidado con esta orden, ya que elimina todos los metadatos asociados y puede resultar peligrosa.

```
test> db.pruebas.drop()
```

2.6. Actualización de documentos

Para la actualización de documentos, podemos optar, bien por actualizaciones de remplazo, mediante el método **`replaceOne()`**, o bien por realizar modificaciones sobre los documentos existentes, mediante los métodos **`updateOne()`**, **`updateMany()`** y **`findOneAndUpdate()`**. Estos métodos recibirán dos argumentos: el primero será el criterio que deberán cumplir los documentos que se quieren actualizar, y el segundo será un documento, bien con el nuevo documento, bien con las actualizaciones que se quieren aplicar.

A. Actualización de reemplazo *(REPLACE)*

La operación de reemplazo, como su nombre indica, **reemplaza todo un documento que cumpla el criterio de actualización por otro documento nuevo**.
Por ejemplo, creamos una nueva colección ***agenda***, para guardar contactos, con información sobre teléfonos:

```
test> db.agenda.insertOne({nombre:"Jose",
telefonos:[{trabajo:"55512345", casa:"555111222"}]}
)
{
  acknowledged: true,
  insertedId: ObjectId("627783dbb54867b80b742df8")
}
```

Como vemos, este método nos devuelve el **_id** del objeto, mediante el cual vamos a poder identificar inequívocamente este documento. Así pues, podríamos reemplazar este documento por otro mediante:

```
test> db.agenda.replaceOne({"_
id":ObjectId("62778439b54867b80b742df9")},
{nombre: 'Jose', correos:[{trabajo: "jose@empresa.com"},
{personal:"jose@proveedor.com"}]} )
{
  acknowledged: true,
  insertedId: null,
   matchedCount: 1,
   modifiedCount: 1,
   upsertedCount: 0
}
```

Como podemos ver, se trata de reemplazar el documento completo, con lo que podemos incluso modificar la estructura de este.

B. Actualizaciones

Las modificaciones se realizan mediante los métodos **updateOne()**, **updateMany()** y **findOneAndUpdate()**. De forma similar a las operaciones de borrado, el método **updateOne()** modificará solamente el primer documento que coincida con el criterio dado, y el método **updateMany(),** todos los que cumplan el criterio. Por su parte, el método **findOneAndUpdate()** modifica el documento y devuelve, por defecto, el documento original, aunque esto es configurable mediante opciones.
Supongamos que tenemos una colección llamada **agenda** que almacena contactos con información sobre teléfonos. Queremos actualizar el número de teléfono de un contacto específico con el nombre "Jose".

1. **updateOne():**

```
db.agenda.updateOne(
  { "nombre": "Jose" }, // Criterio de búsqueda
  { $set: { "telefonos.trabajo": "555555555" } } // Modificación a realizar
)
```

Este código modificará solo el primer documento que coincida con el criterio de búsqueda. En este caso, actualizamos el número de teléfono del trabajo.

2. **updateMany():**

```
db.agenda.updateMany(
  { "nombre": "Jose" }, // Criterio de búsqueda
  { $set: { "telefonos.casa": "666666666" } } // Modificación a realizar
)
```

Este código modificará todos los documentos que coincidan con el criterio de búsqueda. Aquí, actualizamos el número de teléfono de casa para todos los contactos con el nombre "Jose".

3. **findOneAndUpdate():**

```
db.agenda.findOneAndUpdate(
  { "nombre": "Jose" }, // Criterio de búsqueda
 { $set: { "telefonos.personal": "777777777" } }, // Modificación a realizar
  { returnOriginal: false } // Opción para devolver el documento modificado en lugar del original
)
```

Este código modificará el primer documento que coincida con el criterio de búsqueda y devolverá el documento modificado. En este caso, agregamos un nuevo número de teléfono personal al contacto con el nombre "Jose".

C. Modificadores

Los modificadores son claves especiales que **nos permiten especificar operaciones de actualización más complejas.** Normalmente, no necesitaremos reemplazar todo el documento, como en el caso anterior, sino añadir o modificar campos concretos.

En la siguiente tabla vamos a ver los diferentes **modificadores** que tenemos a nuestra disposición:

MODIFICADOR	DESCRIPCIÓN	EJEMPLO DE SINTAXIS (INDISTINTAMENTE CON UPDATEONE O UPDATEMANY)
$set	Asigna valor a un campo al documento. Si este no existe, lo crea. `db.coleccion.updateOne({criterio}, {$set: {campo:valor} });`	
$unset	Elimina un campo de uno o varios documentos. Dado que debemos introducir un par clave-valor, añadiremos un booleano como valor.	`db.coleccion.updateMany({criterio}, {$unset: {campo:true} });`
$inc	Incrementa o decrementa el valor numérico de una clave (no se refiere al identificador), creando una nueva, si no existe.	`db.coleccion.updateOne({criterio}, {$inc: {campo:incremento} });`
$push	Añade elementos a un vector. Si el vector no existe, lo crea, con los elementos que indicamos en el ***push***, mientras que, si ya existe, los añade al final de este.	`db.coleccion.update({criterio}, {$push: {nombre_array:{lista_de_valores} } });`
$pull	Elimina elementos de un vector de acuerdo con algún criterio.	`db.colleccion.update({criterio}, {$pull:{vector:elemento}})`
$pop	Elimina elementos de un vector tratado como una pila o cola, es decir, eliminando el primer (–1) o el último (1) elemento.	`db.colleccion.update({criterio}, {$pop:{vector: [ -1 | 1 ] }})`

D. UPSERTS

Cuando no se encuentra ningún documento que coincida con los criterios de una actualización, como es de esperar, no se produce ninguna modificación en la colección.
En cambio, en ocasiones, podemos desear que, si un documento con ciertos criterios no existe cuando queremos modificarlo, este se cree. Esto se consigue mediante **`updates`** especiales, denominados **`upserts`**. Con esta operación, nos ahorramos buscar primero en la colección, para saber si tenemos que realizar una operación de inserción (si no existe) o de modificación (si existe).
Para realizar un **`upsert`**, utilizaremos el tercer argumento de los **`updates`**, que consiste en un documento con diferentes opciones en formato clave-valor, añadiendo la clave **`upsert`** a valor **`true`**.

```
db.coleccion.updateOne({criterio},{modificación}, {upsert:true});
```

03 Consultas con MongoDB

3.1. El comando *find()*

La orden **`find`** nos permite **recuperar los documentos de una colección que coincidan o hagan match con un criterio especificado** como un documento JSON.
Su sintaxis básica es la siguiente:

```
db.coleccion.find({criterio_en_formato_JSON});
```

Debemos tener en cuenta aspectos como que los tipos de datos que utilicemos sí son una cuestión importante, ya que no es lo mismo el documento **`{edad:20}`** que **`{edad:"20"}`**. Por otro lado, hay que considerar también que el documento vacío **`{}`** casa con todos los documentos, de forma que la siguiente consulta devolvería todos los objetos de la colección:

```
db.coleccion.find({});
```

A. Especificar las claves que hay que recuperar

El comando **`find`** **proporciona los documentos completos que coincidan con el criterio de selección.** Si no deseamos obtener todas las claves, podemos especificar qué claves deseamos consultar, incorporándolas en un segundo parámetro:

```
db.coleccion.find({documento_de_consulta}, {clave_1:1, clave_2:1});
```

Como vemos, este segundo parámetro también se expresa en formato JSON y está compuesto por dos claves (**`clave_1`** y **`clave_2`**), ambas con valor **`1`**. Este valor numérico se interpreta también con el valor **`true`**. Es decir, especificamos aquí cuáles son los campos que deseamos mostrar. En caso de que deseemos mostrar todos los campos y ocultar algunos, utilizaríamos la misma sintaxis, pero empleando ahora un 0 para aquellos campos que queramos ocultar.

B. Operaciones de comparación

MongoDB nos **permite realizar comparaciones con los datos de tipo numérico**, utilizando siempre el formato de documento JSON:

```
db.coleccion.find({clave: {$operador:valor} });
```

Los operadores de comparación que podemos utilizar en MongoDB son:

OPERADOR DE COMPARACIÓN	SIGNIFICADO
$lt	Menor que
$lte	Menor o igual que
$gt	Mayor que
$gte	Mayor o igual que

C. La operación *OR*

Si deseamos realizar un **filtrado o consulta** donde se cumplan varias condiciones (una operación **AND**), no tendremos más que separar estas por comas en el mismo documento JSON que utilicemos como criterio. En cambio, si lo que deseamos es realizar una operación **OR**, deberemos utilizar algún operador especial.

D. Operadores *$IN* y *$NIN*

Un caso especial de ***OR*** es cuando queremos **comprobar si un campo se encuentra dentro de un conjunto de valores concreto.** Es decir, si es **uno** u **otro** valor. Para ello, utilizamos el operador **`$in`**, de la siguiente forma:

```
db.coleccion.find({clave:{$in:[vector_de_valores]}})
```

Del mismo modo, existe el operador **`$nin`** (**`not in`**), que obtiene los documentos, donde el valor especificado no se encuentra en la lista. Debemos tener en cuenta que en este último caso también se mostrarán aquellos documentos que tengan para la clave el valor a **`null`**.

E. El operador *$OR*

Cuando la operación **OR** la queremos realizar sobre diferentes campos del documento, usaremos el operador **$OR**, al que le pasamos un vector de posibles condiciones, de la siguiente forma:

```
db.coleccion.find({$or:[condicion1, condicion2,...]})
```

F. El operador *$NOT*

El operador **`$NOT`** es un operador **metacondicional**, es decir, se aplica siempre sobre otro criterio, invirtiendo su valor de certeza.
Su sintaxis sería:

```
db.coleccion.find({clave:{$not: {criterio}}}).pretty();
```

G. El operador *$EXISTS*

Recordemos que, **en MongoDB, los documentos no poseen una estructura o esquema común,** por lo que es posible que existan claves definidas solamente en algunos de ellos.
El operador **`$exists`** se utiliza para comprobar la existencia o no de determinada clave.
La sintaxis para utilizar sería:

```
db.coleccion.find({clave:{ $exists: true|false }})
```

Con ello, obtenemos los documentos para los cuales la clave existe o no, según hayamos indicado **`true`** o **`false`** en la consulta.

3.2. Consideraciones sobre los tipos de datos en las consultas

Los tipos de datos en MongoDB pueden tener algunos comportamientos especiales. Vamos a ver algunos casos, para saber qué hacer en determinadas situaciones.

A. Valores nulos

El valor **`null`** casa con las siguientes situaciones:

- Cuando el valor de la clave es **`null`**.
- Cuando la clave no existe en el documento (en este caso, se suele decir que no tiene informado el campo).

B. Expresiones regulares y cadenas de caracteres

Cuando aplicamos un filtrado de documentos por un campo de texto, es posible que no sepamos exactamente el valor del campo por el que deseamos filtrar. **Las expresiones regulares ofrecen un mecanismo muy potente para buscar coincidencias en cadenas de caracteres.**
MongoDB nos permite utilizar estas expresiones de varios modos, bien utilizando expresiones regulares de JavaScript o bien mediante el operador **`$regex`**, que utiliza las expresiones regulares compatibles con Perl (PCRE).

C. Expresiones regulares de JavaScript

Las expresiones regulares de JavaScript se expresan mediante la siguiente sintaxis:

```
{ clave: /patrón/<opciones> }
```

Como vemos, utilizamos un patrón de forma similar a una cadena de texto, pero usando la barra `/` como delimitador, en lugar de las comillas.

D. Expresiones regulares con *$REGEX*

Por su parte, si usamos el operador **`$regex`**, podemos usar las siguientes sintaxis:

```
{ clave: { $regex: /patrón/, $options: '<opciones>' } }
{ clave: { $regex: 'patrón', $options: '<opctiones>' } }
{ clave: { $regex: /patrón/<opciones> } }
```

E. Opciones para expresiones regulares

OPCIÓN	DESCRIPCIÓN	EJEMPLOS
`i`	Las coincidencias son ***case-insensitive***.	`{nombe:/juan/i}` `{nombre: { $regex: 'juan', $options: 'i'} }`
`m`	Permite incluir caracteres como `^` o `$`, para hacer ***matching*** al principio o al final, en cadenas con múltiples líneas.	`{nombe:/^Juan/m}` `{nombre: { $regex: 'Juan', $options: 'm'} }`

x	Ignora espacios en blanco en el patrón de **$regex**, siempre que no se hayan escapado o incluido en una clase de tipo carácter.	**{nombre: { $regex: ' J ua n', $options: 'm'} }**
s	Permite el carácter ***punto*** (**.**) para representar ***cualquier*** carácter, incluido el carácter de nueva línea.	**{nombe:/ju.n/s}** **{nombre: { $regex: 'ju.n', $options: 's'} }**

Puedes encontrar más información relativa a las expresiones regulares y casos particulares en que se recomienda usar un tipo de expresión u otra en la documentación oficial de MongoDB acerca de **$regex**.

F. Consultas con vectores

Para **buscar elementos coincidentes dentro de un vector**, procedemos con la misma sintaxis que si se tratase de cualquier otra clave, mediante el documento de consulta **{clave:valor}**, de forma que la clave es un vector, y el valor, bien **n** valores que debe contener el vector, o bien otro vector ordenado con el que queramos que coincida de forma exacta.
Por ejemplo:

- **db.coleccion.find({ mi_vector : valor })**
  Coincide con todos los documentos en cuyo vector **mi_vector** aparezca, en la posición que sea el valor indicado.
- **db.coleccion.find({ mi_vector : [valor] })**
  Coincide con todos los documentos en cuyo vector **mi_vector** aparezca únicamente el valor indicado.

Además, podemos utilizar como condiciones también expresiones regulares, o el resto de operadores que hemos visto.
Por otro lado, también podemos hacer referencia a un elemento concreto del vector por su índice, utilizando la notación punto y entre comillas:

```
db.coleccion.find({ "mi_vector.posicion" : [valor] })
```

G. El operador *$ALL*

Con **$all** podemos especificar más de un elemento coincidente dentro del vector:

```
db.coleccion.find({ mi_vector : {$all:[valor1, valor2, ...]} })
```

H. El operador *$SIZE*

Mediante **`$size`** podemos incluir condiciones sobre la longitud de los vectores:

```
db.coleccion.find({ mi_vector : {$size:tamaño} })
```

I. El operador *$SLICE*

El operador **`$slice`** nos permite obtener un subconjunto de elementos del vector, con la siguiente sintaxis:

OPCIÓN	CUANTITATIVAS
clave: {$slice: x}	Si *x* > 0, obtiene los x primeros elementos. Si *x* < 0, obtiene los últimos x elementos.
clave: {$slice: [x , y] }	Obtiene y elementos a partir del elemento en posición *x*.

J. Documentos incrustados

Para **realizar consultas sobre documentos incrustados**, solo hay que especificar la ruta completa de claves, entrecomillada y separada por puntos:

```
db.coleccion.find({"ruta.a.la.clave":valor_o_condicion})
```

3.3. Cursores

Cuando realizamos una consulta, MongoDB nos devuelve los resultados mediante cursores, que son punteros a los resultados de la consulta. Los clientes que utilizan Mongo son quienes iteran sobre estos cursores para recuperar los resultados, y ofrece un conjunto de funcionalidades, como limitar los resultados, etc.
Cuando realizamos alguna consulta sobre una base de datos con muchos resultados, el cliente nos retorna únicamente 20 resultados y el mensaje **`Type "it" for more`**, para seguir ***iterando*** el cursor.

LIMIT, SKIP Y SORT

MongoDB nos permite realizar ciertas limitaciones sobre los resultados. Entre ellas, podemos destacar:

- **`límite`**: para limitar el número de resultados.
- **`skip`**: salta un número de resultados concreto.
- **`sort`**: ordena los resultados. Necesita un objeto JSON con las claves para ordenar, y un valor 1 para ordenar de forma ascendente, o –1, descendente.

3.4. Introducción al Aggregation Framework

Las consultas de agregación que realizábamos con operadores como **`GROUP BY`**, **`SUM`** o **`COUNT`** en SQL, pueden realizarse con el **`Aggregation Framework`** de MongoDB. Las consultas agregadas tienen la siguiente sintaxis:

```
db.coleccion.aggregate( [<pipeline>] )
```

El **`pipeline`** o tubería tiene un concepto similar a las tuberías de **Unix:** se pasan los resultados de una orden como entrada a otra, para obtener resultados de forma conjunta. Las operaciones que podemos realizar dentro de estas consultas agregadas son:

`$project:`	Para realizar una proyección sobre un conjunto de datos de entrada, añadiendo, eliminando o recalculando campos para que la salida sea diferente.
`$match:`	Filtra la entrada para reducir el número de documentos, dejando solo aquellos que cumplen ciertas condiciones.
`$límit:`	Restringe el número de resultados.
`$skip:`	Ignora un número determinado de registros.
`$unwind:`	Convierte un vector para retornarlo separado en documentos.
`$group:`	Agrupa documentos según determinada condición.
`$sort:`	Ordena un conjunto de documentos, según el campo especificado.
`$geoNear:`	Se usa como datos geoespaciales, y devuelve los documentos ordenados por proximidad según un punto geoespacial.

Para realizar cálculos sobre los datos producidos por las tuberías, usaremos expresiones. Las expresiones son funciones que realizan cierta operación sobre un grupo de documentos, vector o campo concreto. Algunas de estas expresiones son **`$max`**, **`$min`**, **`$divide`** o **`$substr`**.

Puedes encontrar mucha más información sobre Aggregation Framework en la documentación oficial de MongoDB.

Supongamos que tenemos una colección llamada **`ventas`** que almacena información sobre las ventas realizadas en una tienda. Vamos a realizar algunas consultas de agregación para obtener información útil a partir de estos datos.

1. **Proyección y filtro:**

```
db.ventas.aggregate([
  { $match: { fecha: { $gte: ISODate("2024-01-01"), $lte: ISODate("2024-12-31") } } }, // Filtramos por un rango de fechas
  { $project: { _id: 0, producto: 1, cantidad: 1, total: { $multiply: ["$precio", "$cantidad"] } } } // Proyectamos solo los campos necesarios y calculamos el total de cada venta
])
```

Esta consulta filtra las ventas realizadas durante el año 2024 y proyecta solo el nombre del producto, la cantidad vendida y el total de cada venta.

2. **Agrupación y sumarización:**

```
db.ventas.aggregate([
  { $group: { _id: "$producto", totalVentas: { $sum: "$cantidad" } } } // Agrupamos las ventas por producto y sumamos la cantidad vendida de cada uno
])
```

En esta consulta, agrupamos las ventas por producto y sumamos la cantidad vendida de cada uno para obtener el total de ventas por producto.

3. **Ordenamiento y límite:**

```
db.ventas.aggregate([
  { $group: { _id: "$cliente", totalCompras: { $sum: "$total" } } }, // Agrupamos las ventas por cliente y sumamos el total de sus compras
  { $sort: { totalCompras: -1 } }, // Ordenamos los resultados por el total de compras en orden descendente
  { $limit: 5 } // Limitamos los resultados a los 5 clientes con mayores compras
])
```

Esta consulta agrupa las ventas por cliente, calcula el total de compras de cada uno, los ordena de mayor a menor por el total de compras y finalmente muestra solo los 5 clientes con mayores compras.

04 MongoDB y Java

4.1. Drivers

Para conectarnos desde nuestras aplicaciones a una base de datos necesitamos de un controlador o **driver.**
MongoDB ofrece controladores oficiales para multitud de plataformas, entre las que encontramos C, C++, C#, NodeJS, Python, y como no, Java, entre muchas otras.
Centrándonos en Java, MongoDB nos ofrece dos **drivers:**

- El ***driver*** de Java para aplicaciones síncronas.
- El ***driver*** Reactive Streams para el procesamiento de **streams** asíncronos.

Aunque actualmente se tiende a la programación reactiva, vamos a trabajar con el ***driver*** síncrono de Java para facilitar la comprensión y centrarnos en el acceso propiamente dicho a los datos.

A. El driver de Java

Mediante el driver de MongoDB para Java podemos conectarnos tanto a una base de datos local o remota como a un clúster de MongoDB Atlas.
Este **driver** (MongoDB Java Driver) lo podemos encontrar en los repositorios de Maven, y proporciona una gran cantidad de clases e interfaces para facilitar el trabajo con MongoDB desde Java.

DRIVER DE MONGODB PARA JAVA	DESCRIPCIÓN
Nombre	MongoDB Java Driver
Versión	Variable, consulta la documentación para la última versión disponible
Plataformas Soportadas	Java

Conexión	Permite conectarse tanto a una base de datos MongoDB local como a un clúster de MongoDB Atlas
Repositorios	Disponible en los repositorios de Maven para una fácil integración en proyectos Java
Funcionalidades	»Realiza operaciones CRUD (Crear, Leer, Actualizar, Borrar) en documentos MongoDB »Ejecuta consultas de agregación para análisis avanzado de datos »Manipula índices y realiza operaciones de indexación para optimizar consultas »Ofrece una gran cantidad de clases e interfaces diseñadas para facilitar el trabajo con MongoDB desde aplicaciones Java

B. Conexión a una base de datos

Para conectarnos y comunicarnos con una base de datos necesitamos un cliente. En el caso del **driver** de Java para MongoDB, el cliente se implementa mediante la clase MongoClient.

LA CLASE MONGOCLIENT

La clase `MongoClient` representa un conjunto de conexiones a un servidor MongoDB. Estas conexiones son ***thread-safe***, es decir, que varios hilos de ejecución pueden acceder de forma segura a ellas.
La forma de crear instancias de **MongoClient** es mediante el método `MongoClients.create()`. Además, generalmente, solo necesitaremos una instancia de esta clase, incluso en aplicaciones multihílo.
El método `create` de **MongoClients** toma como argumento una cadena de conexión (Connection String), con el siguiente formato simplificado (los parámetros más claros son opcionales):

```
mongodb:// [usuario:contraseña @] host[:puerto] /?opciones
```

Así pues, una forma de obtener, por ejemplo, una conexión al servidor local sería:

```
String uri = "mongodb://localhost:27017";
MongoClient mongoClient = MongoClients.create(uri);
```

La clase `MongoClient`, entre otros, admite los métodos siguientes:

MÉTODO	SIGNIFICADO
getDatabase(String nombre)	Obtiene una referencia a una base de datos cuyo nombre le pasamos como argumento.
listDatabaseNames()	Obtiene una lista de strings (interfaz MongoIterable) con los nombres de las bases de datos del servidor.

close()	Cierra la conexión al servidor. Debe realizarse siempre cuando ya no se vaya a utilizar.

En el enlace **Connect to MongoDB**, de la sección correspondiente, dispones de más información acerca de estos y otros métodos de crear una conexión al servidor.

MONGODATABASE

El método `getDatabase` de la clase `MongoClient` nos retorna una referencia a un objeto que implementa la interfaz `MongoDatabase`, que representa una conexión a una base de datos.
Esta interfaz define los siguientes métodos:

MÉTODO	SIGNIFICADO
getCollection(String nombre)	Obtiene una referencia a la colección.
listCollectionNames()	Obtiene una lista de ***strings*** (interfaz **MongoIterable**) con los nombres de las colecciones en la base de datos.
listCollections()	Obtiene una lista de referencias (**MongoCollection**) a las colecciones de la base de datos.
createCollection(String nombre)	Crea una nueva colección con el nombre indicado en la base de datos.
drop()	Elimina la base de datos.

En el caso práctico de este apartado, veremos cómo utilizar algunos de estos métodos.

C. Consultas

El método `getCollection()` de `MongoDatabase()` nos proporciona una colección de documentos, sobre los que vamos a poder realizar consultas mediante el método `find()`. Este método, que ya conocemos de la `shell` de MongoDB, nos permitirá realizar un filtrado de documentos de acuerdo con ciertos criterios.
Estos criterios se expresan como `filtros` (`query filters` en la documentación), y pueden contener varios operadores de consulta sobre algunos campos, que determinarán qué documentos de la colección se incluyen como resultados.
La clase `Filter` nos proporciona métodos de factoría para realizar estas consultas, de forma similar a como trabajábamos con la ***shell*** de MongoDB. Esta clase nos proporciona:

- **Consulta vacía**, con `Filters.empty()`.
- **Operadores de comparación** para realizar consultas de acuerdo con valores en la colección:

  `Filters.eq(clave, valor)`, `Filters.gt(clave, valor)`, `Filters.gte(clave, valor)`, `Filters.lt(clave, valor)` ***o*** `Filters.lte(clave, valor)`.

- **Operadores lógicos** para realizar operaciones lógicas sobre el resultado de otras consultas:

  `Filter.and(otros_filters), Filter.or(otros_filters)...`

- **Operadores de arrays.** Nos permiten realizar consultas de acuerdo con el valor o la cantidad de elementos de un vector: `Filters.size(vector, tamaño).`
- **Otros operadores,** como `Filter.exists()` o `Filter.regex()`, para comprobar la existencia de una clave o bien realizar una búsqueda por expresiones regulares.

Supongamos que tenemos una colección llamada `empleados` que almacena información sobre los empleados de una empresa. Vamos a realizar una consulta para encontrar todos los empleados que tengan un salario mayor o igual a $50000.

```
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Filters;
import org.bson.Document;

public class Main {
    public static void main(String[] args) {
        // Conexión a la base de datos
        try (MongoClient mongoClient = MongoClients.create("mongodb://
localhost:27017")) {
            MongoDatabase database = mongoClient.getDatabase("mi_base_
de_datos");
                    MongoCollection<Document> collection = database.
getCollection("empleados");

            // Construcción de la consulta con Filter
            Document queryFilter = Filters.gte("salario", 50000);

            // Realización de la consulta y obtención de los resultados
            collection.find(queryFilter).forEach(System.out::println);
        }
    }
}
```

En este ejemplo, utilizamos el método **`getCollection()`** para obtener una instancia de **`MongoCollection`**, sobre la cual podemos realizar la consulta utilizando el método **`find()`**. Construimos el filtro de consulta utilizando la clase **`Filters`**, específicamente utilizando el método **`gte()`** para especificar que el salario debe ser mayor o igual a $50000. Luego, ejecutamos la consulta y recorremos los resultados para imprimirlos

Además de los filtros, también vamos a poder incluir operaciones de agregación, mediante el método **`aggregate()`** de una instancia de MongoCollection. Puedes consultar la documentación acerca de las agregaciones en la guía de operaciones de agregación de MongoDB, referenciada en los enlaces.

Por otra parte, la API del ***driver*** de MongoDB también nos permite realizar proyecciones de campos mediante la clase **`Projections`**, que ofrece los métodos **`Projections.fields()`**, **`Projections.include()`** o **`projections.excludeID()`**.

4.2. Spring Data MongoDB y API REST

Como sabemos, el proyecto Spring Data, incluido en la plataforma Spring, proporciona un ***framework*** para simplificar el acceso a datos y la persistencia sobre diferentes repositorios de información.

Dentro de este proyecto se encuentra Spring Data MongoDB, que **proporciona integración con bases de datos MongoDB, mediante un modelo centrado en POJO**, que interactúan con las colecciones de documentos y proporcionan un repositorio de acceso a datos.

En este apartado, y tomando el relevo de la unidad anterior, vamos a abordar el desarrollo de componentes de acceso a datos mediante Spring Data, así como microservicios que ofrezcan estos datos mediante una API REST; todo ello, siguiendo el patrón MVC que ya conocemos.

A. Definición del modelo-documento

Una base de datos en MongoDB está compuesta por colecciones de documentos. Aunque estos documentos puedan tener diferentes estructuras entre sí o diferentes tipos de datos, el modelo sí que requiere de una estructura estática.

Así pues, lo primero que debemos hacer es crear una clase que represente este documento principal para MongoDB, que será el devuelto por las consultas que se vayan realizando.

En este contexto, son dos las principales anotaciones que utilizaremos:

- **`@Document`**, para indicar que una clase se corresponde con un objeto de dominio (**`domain object`**), que puede mapearse en la base de datos para ofrecer persistencia. Esta anotación para MongoDB sería el equivalente a **`@Entity`** en JPA.

Si no se indica nada, se interpretará que el nombre de la colección que se va a utilizar se corresponde con el nombre de la clase en minúscula. Así pues, si tenemos la clase **`com.mgh.ad.Persona`**, se utilizará la colección «persona». No obstante, podemos indicar la colección con la que trabajamos, mediante los atributos **`value`** o **`collection`**, con las siguientes sintaxis:

- DirectorioA
- **`@Document(value="coleccion")`**
- **`@Document("coleccion")`**
- **`@Document(collection="coleccion")`**

» **`@Id`**: se aplica sobre un campo, y sirve para indicar que el campo se utilizará como identificador. Como sabemos, todo documento en MongoDB requiere de un identificador. Si no se proporciona uno, el controlador asignará un **`ObjectID`** automáticamente. Es importante destacar que los tipos de datos que podemos utilizar como identificadores pueden ser tanto ***strings*** como **`BigInteger`**, ya que Spring se encargará de convertirlos al tipo **`ObjectID`**.

Además de estas, existen otras anotaciones más específicas que podemos utilizar. Si lo deseas, puedes consultarlas en la documentación de referencia de Spring Data MongoDB.

B. Definición del repositorio

Como sabemos, el repositorio es la interfaz encargada de gestionar el acceso a los datos. Para el caso de MongoDB, esta derivará de **`MongoRepository`**, que será una interfaz parametrizada por dos argumentos:
`MongoRepository<T, Id>:`

» **`T`**: el tipo de documento, que se corresponderá a la clase definida en el modelo, e

» **`Id`**: el tipo de datos al que pertenecerá el identificador.

La interfaz **`MongoRepository`**, como hemos dicho, será específica para MongoDB, y derivará de las interfaces **`CrudRepository`** y **`PagingAndSortingRepository`**, de las que heredará todos sus métodos.
De este modo, en el repositorio únicamente deberemos declarar aquellos métodos que sean más específicos para nuestra aplicación, ya que todos los métodos para implementar operaciones CRUD, así como **`findAll()`** y **`findById()`** los heredará de **`MongoRepository`**.
Para definir en el repositorio nuestras propias consultas, utilizaremos la anotación **`@Query`**, proporcionándole como valor la consulta en cuestión:

```
@Query(value="{ consulta_parametrizada }")
List<TipoDocumento> nombreMetodo(lista_parametros);
```

Para proporcionar parámetros a la consulta, estos se reciben como argumentos del método, y son referenciados por su orden en la consulta: `?0` para el primer argumento, `?1` para el segundo, etc.

C. Definición del servicio

Los servicios se encargan de la capa de negocio de nuestra aplicación, y acceden a los datos a través del repositorio, enviando los resultados al controlador. Estos servicios, en general, se caracterizan por:

- Utilizar la anotación **`@Service`** para indicar a Spring que se está implementando un servicio.
- Por una parte, suele definirse la interfaz Service, y por otra realizarse la implementación mediante la clase **`ServiceImpl`.**
- Se utiliza la anotación **`@Autowired`** en las referencias a los repositorios para enlazar o inyectar el servicio en cuestión con dicho repositorio.
- Una vez que obtiene los datos del repositorio, los envía al controlador.

D. Definición del controlador

Finalmente, nos queda la implementación del controlador, que ya conocemos de Spring. Recordemos las principales características de este:

- Utiliza la anotación **`@RestController`** a nivel de clase para indicar que se trata de un controlador **`REST`**.
- Utiliza la anotación **`@RequestMapping`** a nivel de clase para especificar la ruta base para los **`endpoints`** del servicio.
- Utiliza la anotación **`@Autowired`** en las propiedades que hacen referencia al servicio, para inyectar este de forma automática.
- Utiliza las anotaciones **`@GetMapping`**, **`@PostMapping`**, **`@PutMapping`**, **`@DeleteMapping`** sobre los métodos que implementarán peticiones de tipo **`GET`**, **`POST`**, **`PUT`** o **`DELETE`**, especificando su **`Endpoint`**. • Utiliza la anotación **`@PathVariable`** en los argumentos de los métodos anteriores para obtener las variables a partir de la URL, que especificaremos entre llaves **`{variable}`** en el **`endpoint`.**

Documentación oficial:

- Spring Data MongoDB: https://spring.io/projects/spring-data-mongodb
- Referencia de la API de Spring Data MongoDB: https://docs.spring.io/spring-data/mongodb/reference/index.html

Guías y tutoriales:

- Guía oficial de Spring para MongoDB: https://www.mongodb.com/resources/products/compatibilities/spring-boot
- Tutorial de Spring Boot con MongoDB: https://www.baeldung.com/spring-data-mongodb-guide
- Configuración de Spring Data MongoDB: https://mkyong.com/tutorials/java-mongodb-tutorials/
- Ejemplo de aplicación Spring Boot con MongoDB: https://github.com/rahul-ghadge/spring-boot-mongodb-crud

UAX FP