

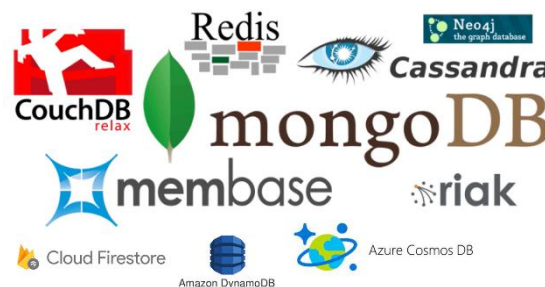
Unidad 4: Bases de datos documentales.

Índice de contenidos

1	¿Qué son las <i>BD NoSQL</i> ?	2
2	Formato <i>JSON</i> .	4
3	<i>MongoDB</i> .	5
3.1	¿Qué es <i>MongoDB</i> ?	5
3.2	Elementos con los que trabaja.	6
3.3	Tipos de datos.	6
3.4	Instalación.	7
3.5	Herramientas gráficas.	9
3.5.1	MongoDB Compass.	9
3.5.2	Studio 3T.	10
3.6	<i>MongoDB Atlas</i> .	10
3.7	<i>BD</i> del sistema.	13
3.8	Comandos básicos (<i>MongoDB Shell</i>).	13
3.9	Colecciones, documentos y <i>BSON</i> .	15
3.10	Operaciones con colecciones.	17
3.11	<i>CRUD</i> de documentos.	18
3.11.1	Inserción.	18
3.11.2	Actualización.	18
3.11.3	Eliminación.	21
3.11.4	Consulta.	21
3.12	Operaciones de consulta.	22
3.12.1	Consultas y arrays.	23
3.12.2	Selección de campos a devolver en las consultas.	24
3.12.3	Consulta de campos nulos o faltantes.	24
3.13	Agregaciones y tuberías.	25
3.13.1	Métodos de agregación de propósito único.	25
3.13.2	Tuberías y transformaciones.	25
3.14	Funciones.	27
3.15	Índices.	28
3.15.1	Método <i>explain</i> .	28
3.15.2	Creación de un índice.	28
3.15.3	Eliminación de un índice.	29
3.16	Modelado de datos.	29
3.17	Operador <i>Lookup</i> .	33
3.18	Aplicar reglas de validación de esquema en una colección.	34
3.19	Usuarios.	35
3.19.1	Activar la autenticación y agregar el usuario <i>root</i> a <i>MongoDB</i> en Windows.	36
3.19.2	Crear un usuario.	36
3.19.3	Cambiar la contraseña de un usuario.	36
3.19.4	Eliminar un usuario.	37
3.19.5	Listar todos los usuarios.	37
3.19.6	Otorgar roles a un usuario.	38
3.19.7	Revocar roles a un usuario.	38
3.20	Transacciones.	38
3.20.1	Ejemplo de transacción.	39
4	Usando <i>MongoDB</i> y <i>Java</i> .	41
4.1	Dependencias.	41
4.2	Conexión a <i>MongoDB</i> .	41
4.3	Pool de conexiones.	43
4.4	Bases de datos, colecciones y documentos.	45
4.5	Trabajar con bases de datos.	45
4.6	Trabajar con colecciones.	47
4.7	Creación de documentos.	48
4.8	Inserción de documentos.	48
4.9	Búsqueda de documentos.	49
4.9.1	Filters.	52
4.9.2	Ordenación, proyección, <i>skip</i> y <i>limit</i> .	53
4.9.3	Colaciones.	54
4.10	Eliminación de documentos.	55
4.11	Actualización de documentos.	56
4.12	Framework de agregación.	61
4.13	Acceder a una colección con un <i>POJO</i> .	63

1 ¿Qué son las BD NoSQL?

Las **BD NoSQL** (*Not Only SQL*) se distinguen de las **BD** relacionales en su **diseño y enfoque al no adoptar el modelo de tablas y relaciones**. En lugar de ello, optan por una **estructura de almacenamiento más flexible y escalable**, empleando diversos modelos de datos no relacionales, como **documentos**, **grafos**, datos geoespaciales, **clave-valor** y columnas.



El término **NoSQL** surgió con la popularización de la web 2.0. Hasta ese momento, solo las empresa con portales web subían contenidos a *Internet*. Sin embargo, con la llegada de aplicaciones como *Facebook*, *Twitter* o *YouTube*, en las cuales los usuarios interactúan directamente en la web, cualquier persona podía subir contenido. Esto generó un crecimiento exponencial de los datos, impulsando la necesidad de nuevas soluciones de almacenamiento de datos más flexibles y escalables, lo que condujo al desarrollo de las **BD NoSQL**.

Hablar de **BD NoSQL** implica referirse a estructuras diseñadas para almacenar información en situaciones donde las **BD** relacionales enfrentan dificultades, especialmente en términos de escalabilidad y rendimiento. Este desafío se manifiesta especialmente en entornos con una gran cantidad de usuarios simultáneos y un elevado número de consultas diarias. Por lo tanto, las **BD NoSQL** **buscan abordar problemas relacionados con el almacenamiento a gran escala, el rendimiento óptimo y el procesamiento masivo de transacciones**, particularmente en sitios web con un tráfico considerable. En resumen, estas **BD** representan una alternativa **NoSQL** para la gestión y persistencia eficiente de grandes volúmenes de datos en las organizaciones.

Las **BD NoSQL** han ganado popularidad gracias a su capacidad para gestionar grandes volúmenes de datos no estructurados y semiestructurados, como los datos procedentes de redes sociales y los registros de aplicaciones web. Además, estas **BD** se integran eficazmente con la nube y ofrecen escalabilidad horizontal, lo que significa que pueden expandirse fácilmente añadiendo más servidores o nodos a medida que crece tanto la cantidad de datos como la carga de trabajo.

Por ejemplo, considerar el escenario donde se desea modelar el esquema de una **BD** sencilla para almacenar información sobre libros. Si se opta por una **BD** relacional, se organizaría la información en distintas tablas, las cuales estarían interrelacionadas mediante restricciones de claves primarias y foráneas. Este modelo relacional está concebido para mantener la integridad referencial entre las tablas y minimizar la redundancia de la información a través de procesos de normalización. En este caso específico, se podrían tener las siguientes tablas:

```
LIBRO(isbn, título, año_edición)
AUTOR(id_autor, nombre_autor)
ESCRIBE(id_autor[fk: AUTOR], isbn[fk: LIBRO])
```

En una **BD NoSQL**, el registro de un libro con su autor se podría almacenar en un único documento **JSON**:

```
{
  "isbn": 9788448190330,
  "título": "Fundamentos de Bases de Datos",
  "año_edición": 2014,
  "id_autor": 11,
  "nombre_autor": "Abraham Silverschatz"
}
```

Las **BD** relacionales se centran en garantizar la fiabilidad de las transacciones mediante los principios **ACID** (atomicidad, consistencia, aislamiento y durabilidad):

- ✓ **Atomicity**: un cambio debe realizarse en su totalidad o no efectuarse en absoluto.
- ✓ **Consistency**: cualquier cambio debe llevar la **BD** de un estado válido a otro estado válido según las restricciones y el esquema de datos establecidos.
- ✓ **Isolation**: un cambio no debe interferir con otros cambios que se estén ejecutando simultáneamente en la **BD**. Garantiza la independencia entre transacciones.
- ✓ **Durability**: una vez que se ha completado el cambio, este debe mantenerse, incluso si se produce un fallo en la **BD** o en todo el sistema.

Cuando se trata de **grandes volúmenes de datos y una alta dinámica en su gestión**, los **principios ACID de los modelos relacionales pueden generar limitaciones** en cuanto a rendimiento y operatividad. En estos casos, las **BD**

NoSQL destacan al **priorizar** características como el **rendimiento, la disponibilidad y la escalabilidad**, relegando la importancia de los principios **ACID**.

Hoy en día, los **modernos sistemas de datos en Internet** se ajustan más a los principios **BASE**:

- ✓ **Basic Availability**: prioridad en la disponibilidad de los datos.
- ✓ **Soft State**: se prioriza la propagación de datos, dejando el control de inconsistencias a elementos externos.
- ✓ **Eventually Consistency**: se asume que inconsistencias temporales progresarán hacia un estado final estable.

La forma de almacenamiento de información en este tipo de **BD** ofrece ciertas **ventajas sobre los modelos relacionales**:

- ✓ **Se ejecutan en máquinas con pocos recursos**: estos sistemas no requieren mucha programación, lo que permite su instalación en máquinas de menor costo.
- ✓ **Escalabilidad horizontal**: el rendimiento de estos sistemas se mejora simplemente agregando más nodos y notificando al sistema sobre la disponibilidad de esos nodos.
- ✓ **Pueden manejar gran cantidad de datos**: gracias a su estructura distribuida, que en muchos casos se implementa mediante tablas *hash*.
- ✓ **Evitan los cuellos de botella**: en **SQL**, cada sentencia debe ser procesada individualmente antes de su ejecución, lo que puede volverse especialmente complejo con consultas complicadas. Este proceso crea un punto de congestión que, ante un gran número de solicitudes, puede ralentizar significativamente el sistema.

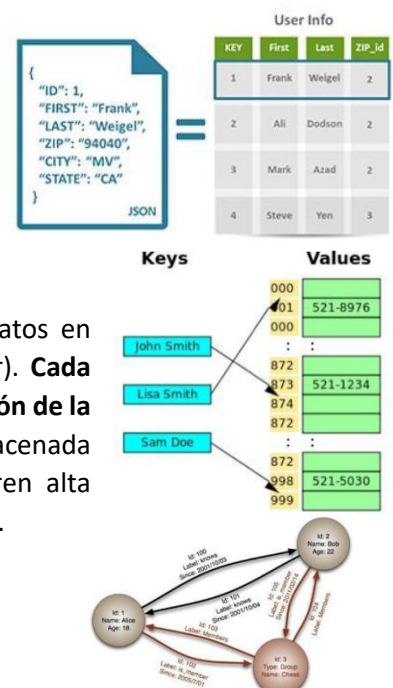
Una de las principales diferencias al migrar a **NoSQL** es que los datos no se almacenan en tablas tradicionales, sino en colecciones de objetos, documentos o pares clave-valor, prescindiendo de un esquema fijo.

Según la forma en la que se organice la información, **es posible encontrarse con distintos tipos de BD NoSQL**. Cada tipo presenta sus propias ventajas y desventajas, por lo que es crucial elegir el tipo adecuado según los requisitos particulares de la aplicación a desarrollar.

Las DB NoSQL **priorizan la velocidad sobre la integridad de los datos** y permiten redundancias para agilizar las consultas.

Los **tipos de BD NoSQL más utilizados** son:

- ✓ **Documentales**: almacena la **información como documentos**, generalmente utilizando para ello una estructura simple como **JSON** y donde se utiliza una clave única para cada registro. Son las **BD NoSQL más versátiles**. **Se pueden utilizar en gran cantidad de proyectos**, incluyendo muchos que tradicionalmente funcionarían sobre **BD relacionales**. Ejemplos de **SGBD documentales**: **MongoDB**, **Amazon DocumentDB**, **Apache CouchDB**, **RavenDB**.
- ✓ **Clave-valor**: es la más sencilla en cuanto a funcionalidad, organizan los datos en parejas de clave (un identificador) y valor (datos asociados al identificador). **Cada elemento está identificado por una clave única, lo que permite la recuperación de la información de forma muy rápida**, información que habitualmente está almacenada como un objeto binario (**BLOB**). Son útiles para aplicaciones que requieren alta velocidad en la lectura y escritura de datos. Ejemplos: **DynamoDB**, **Redis**, **Riak**.
- ✓ **Grafos**: la **información se representa como nodos** (conjunto de datos como un **JSON**) y **aristas** (conexiones entre ellos), de manera que se puede hacer uso de la teoría de grafos para recorrerla. Ejemplos: **Neo4j**, **Amazon Neptune**, **ArangoDB**, **OrientDB**.



Algunas de las **razones que pueden llevar a decantarse por el uso de las BD NoSQL** en lugar de las **SQL** son:

- ✓ Cuando el **volumen de los datos crece muy rápidamente** en momentos puntuales, pudiendo llegar a superar el **Terabyte** de información.
- ✓ Cuando la **escalabilidad de la solución relacional no es viable** tanto a nivel de costes como a nivel técnico.
- ✓ Cuando **se tienen elevados picos de uso del sistema** por parte de los usuarios en múltiples ocasiones.
- ✓ Cuando el **esquema de la BD no es homogéneo**, es decir, cuando en cada inserción de datos la información que se almacena puede tener campos distintos.

Las BD NoSQL son adecuadas para aplicaciones que manejan grandes volúmenes de datos, requieren alta escalabilidad y alta disponibilidad, y necesitan una alta velocidad de procesamiento.

En esta unidad se va a trabajar con las **BD NoSQL** documentales y concretamente haciendo uso para ello de **MongoDB**.

2 Formato JSON.

JSON (JavaScript Object Notation) es un **formato de intercambio de datos** ligero y muy utilizado en la web para transmitir y almacenar información. Fue diseñado para ser **fácil de leer y escribir**, tanto para humanos como para máquinas, y se basa en un subconjunto del lenguaje de programación **JavaScript**. **Surge como alternativa a XML y utiliza convenciones reconocidas por la mayoría de los lenguajes (C, C++, C#, Java, JavaScript, Perl, Python, ...).**

En el formato **JSON**, los **datos se organizan en estructuras de objetos o matrices**. Los **objetos son conjuntos no ordenados de pares clave-valor**, donde cada clave es una cadena de caracteres y cada valor puede ser cualquier tipo de datos válido en **JSON**, incluyendo objetos y matrices. Las **matrices**, por otro lado, **son conjuntos ordenados de valores**, donde cada valor puede ser cualquier tipo de datos válido en **JSON** (incluyendo otros objetos o matrices).

El formato **JSON** es **ampliamente utilizado en la web para intercambiar datos entre aplicaciones cliente y servidor**, y **también se utiliza como formato de almacenamiento de datos en BD NoSQL documentales, como MongoDB**.

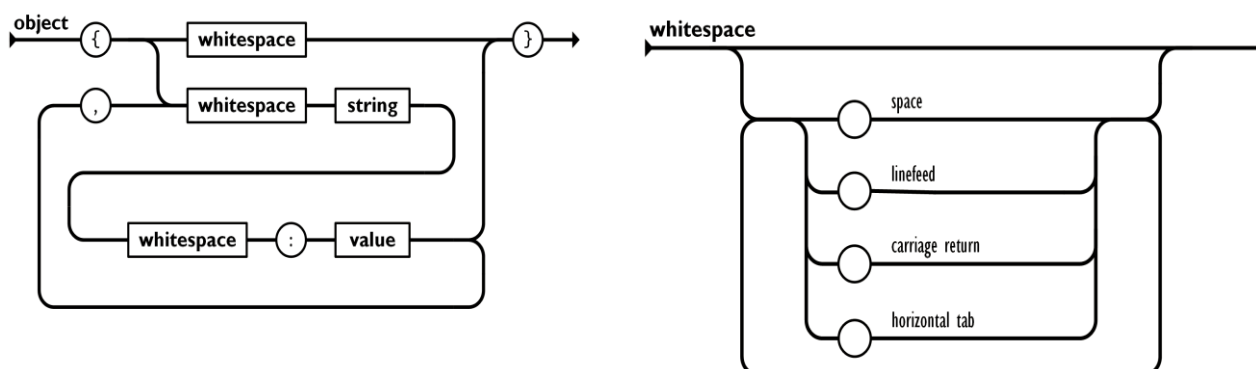
En **JSON**, se pueden representar los siguientes **tipos de datos**:

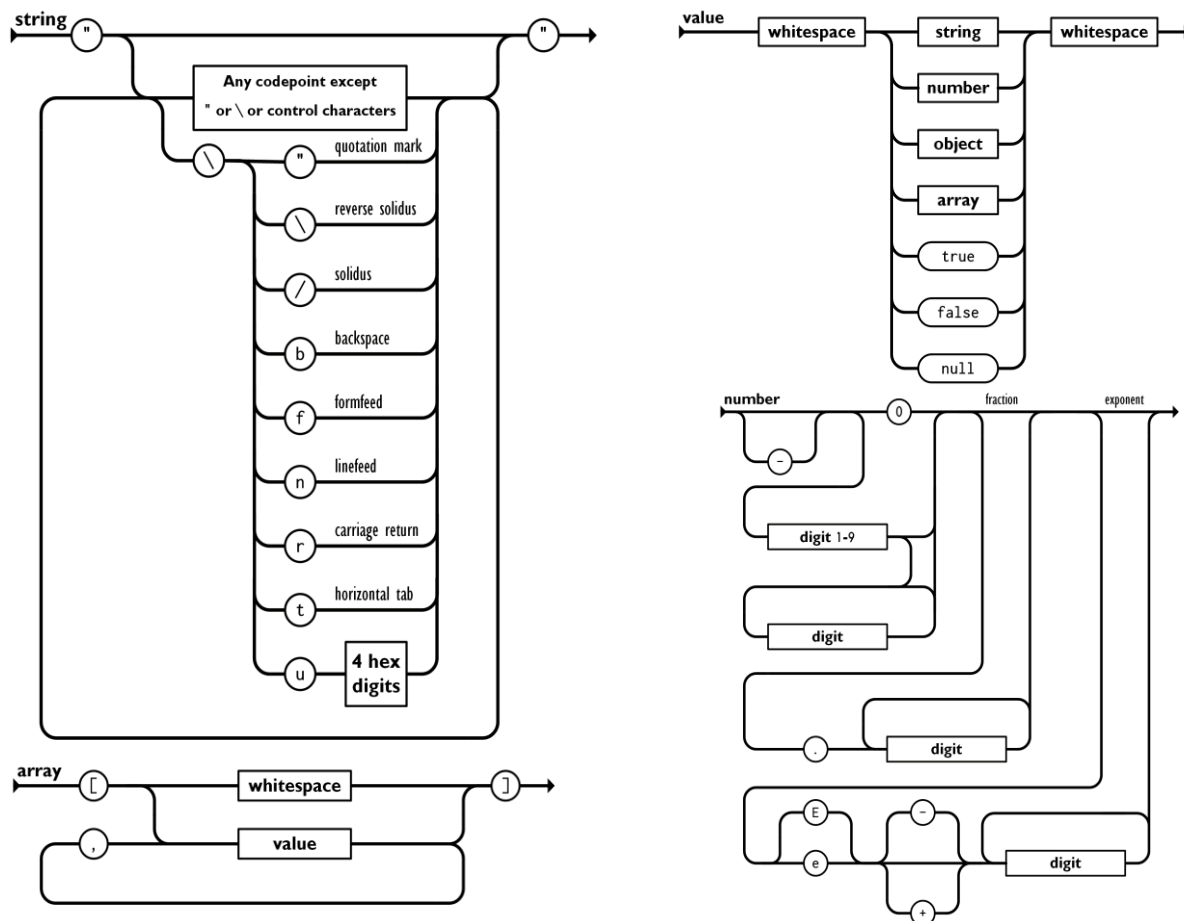
- ✓ **Cadenas de caracteres**: son secuencias de 0 o más caracteres *Unicode* encerradas **entre comillas dobles** que se utilizan **para almacenar texto** (un carácter se considera una cadena de caracteres de un carácter). Ejemplo: "Hola".
- ✓ **Números**: pueden ser **enteros o de punto flotante**, y tomar valores **positivos y negativos**. Ej.: 4 o -3.14.
- ✓ **Booleanos**: son **valores lógicos** que pueden ser verdadero o falso. Ejemplo: **true** o **false**.
- ✓ **Null**: es un valor nulo que **representa la ausencia de un valor**. Ejemplo: **null**.
- ✓ **Arreglos o matrices**: son **conjuntos ordenados de valores** que pueden incluir diferentes tipos de datos. Ejemplo: ["manzana", "naranja", "plátano"].
- ✓ **Objetos**: son **conjuntos no ordenados de pares clave-valor** que representan datos complejos. Ejemplo: {"nombre": "Juan", "edad": 25, "dirección": {"calle": "Av. Central", "ciudad": "Bogotá"}}.

Los **objetos y los arreglos pueden anidarse para representar datos complejos y estructurados**.

Reglas básicas para escribir un documento JSON:

- ✓ Los **objetos** se representan como **conjuntos no ordenados de pares clave-valor**. Cada **par clave-valor** se separa con una coma. El **objeto empieza con llave y termina con llave {}**. La **clave es una cadena de caracteres** y el **valor puede ser de cualquier tipo de datos permitido en JSON**.
- ✓ Los **arreglos** se representan como **conjuntos ordenados de valores**. Cada **valor se separa con una coma**. Un **array comienza con corchete y termina con corchete []**. Los **valores pueden ser a su vez otros objetos**.
- ✓ Los **valores de cadena de caracteres** se encierran entre **comillas dobles (")**.
- ✓ Los **valores numéricos** pueden ser **enteros o de punto flotante** y se escriben **sin comillas**.
- ✓ Los **valores booleanos** se representan como **true** o **false**.
- ✓ El **valor nulo** se representa como **null**.





Ejemplo: documento *JSON* que incluye distintos tipos de datos.

```
{
  "cadena": "Hola, ¡JSON!",
  "numero_entero": 42,
  "numero_flotante": 3.14,
  "booleano": true,
  "nulo": null,
  "array": ["manzana", "naranja", "plátano"],
  "objeto": {
    "nombre": "Juan",
    "edad": 25,
    "dirección": {
      "calle": "Av. Dular",
      "ciudad": "Granada",
      "país": "España"
    }
  }
}
```

Más información en: <http://www.json.org/json-es.html>

Validar si un objeto *JSON* está bien escrito:

- <https://jsonlint.com/>
- <https://jsonformatter.curiousconcept.com/>

En este ejemplo, se incluyen una cadena de caracteres, un número entero, un número de punto flotante, un valor booleano, un valor nulo, un arreglo y un objeto anidado que contiene información sobre una persona. Cada tipo de dato se representa con su sintaxis correspondiente en *JSON* y todos ellos se combinan en un solo documento.

3 MongoDB.

3.1 ¿Qué es MongoDB?



MongoDB es un **SGBD NoSQL** (no relacional) de código abierto, multiplataforma, de alto rendimiento y orientado a documentos que se enfoca en el almacenamiento y procesamiento de datos semiestructurados y no estructurados que nació en 2007 (desarrollado en C++ por la empresa *10gen*, que, en 2013, cambió su nombre a *MongoDB Inc.*) y fue lanzada por primera vez en 2009. Es decir, en lugar de almacenar los datos en tablas, como en las *BD* relacionales, *MongoDB* almacena los datos en **documentos BSON** (*Binary JSON*) que se asemejan a objetos *JSON*.

MongoDB es escalable (desde pequeñas aplicaciones web hasta grandes sistemas empresariales) y **altamente disponible**, lo que lo hace ideal para aplicaciones modernas que requieren un alto rendimiento y una gran cantidad de datos. Además, **MongoDB es muy popular en el mundo de la programación debido a su facilidad de uso y a la gran cantidad de bibliotecas y herramientas disponibles para trabajar con él.**

Entre las **características principales** se encuentran la **capacidad de manejar grandes volúmenes de datos**, la **flexibilidad de su esquema** (permite almacenar cualquier tipo de contenido sin obedecer a un esquema), la **replicación de datos** y la distribución geográfica. También cuenta con una **amplia gama de funcionalidades**, como índices geoespaciales, agregación de datos, búsquedas de texto completo y soporte para transacciones.

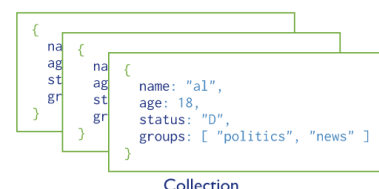
El modelo de documentos de **MongoDB** resulta muy fácil de aprender y usar, y proporciona a los desarrolladores todas las funcionalidades que necesitan para satisfacer los requisitos más complejos. Se proporcionan drivers para más de diez lenguajes de programación, y la comunidad ha desarrollado varias decenas más.

Más información en <https://www.mongodb.com/>

3.2 Elementos con los que trabaja.

MongoDB trabaja con varios **elementos principales**:

- ✓ **Base de datos:** es un **contenedor de colecciones de documentos**. Se pueden tener varias **BD** en un mismo servidor de **MongoDB**, pero cada una debe tener un nombre único.
- ✓ **Colección:** es un **grupo de documentos almacenados en una BD** (cada documento es similar a un registro en una tabla de una **BD** relacional). Las colecciones **no tienen un esquema predefinido** y pueden contener documentos con diferentes estructuras. Cada colección debe tener un **nombre único dentro de la BD** y puede contener un número arbitrario de documentos.
- ✓ **Documento:** es un **objeto JSON que contiene uno o más campos con valores asociados**, que son pares clave-valor. Los documentos de una colección **no tienen que tener la misma estructura**, lo que significa que cada documento puede tener diferentes campos y valores.
- ✓ **Campo:** es una **clave dentro de un documento que se utiliza para almacenar un valor específico**. El valor de un campo puede ser de cualquier tipo de datos compatibles con **BSON** (cadenas de texto, números, booleanos, arreglos, objetos incrustados y otros tipos de datos).
- ✓ **Índice:** es una estructura de datos que **permite acelerar la búsqueda y recuperación de datos**. Los índices se crean en campos específicos de los documentos en una colección.



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

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

3.3 Tipos de datos.

MongoDB admite varios **tipos de datos**, entre los cuales se incluyen:

- ✓ **Cadenas de caracteres:** son **secuencias de caracteres Unicode** que se utilizan para almacenar texto. Ejemplo: "¡Hola, MongoDB!".
- ✓ **Números:** pueden ser **enteros o de punto flotante**. Ejemplos: 42, -10.5 o 3.14.
- ✓ **Booleanos:** son **valores lógicos** que pueden ser verdadero o falso (**true** o **false**).
- ✓ **Arreglos, arrays o matrices:** son **conjuntos ordenados de valores** que pueden incluir diferentes tipos de datos. Ejemplo: ["manzana", "naranja", "plátano"].
- ✓ **Objetos:** son **conjuntos de pares clave-valor que representan datos complejos**. Ejemplo: {"nombre": "Juan", "edad": 25, "dirección": {"calle": "Av. Central", "ciudad": "Bogotá"}}.
- ✓ **ObjectId:** son **identificadores únicos de 12 bytes** (4 bytes marca temporal, 3 bytes id. de máquina, 2 bytes id. de proceso y 3 bytes contador aleatorio). Ejemplo: ObjectId("5f95c64b5e5ca5e55b4f4e4f").
- ✓ **Objetos binarios:** son objetos que **contienen datos binarios**. Ejemplo: BinData(0, "R0lGODlhAQABAIAAAAAAAP///yH5BAEAAAAALAAAAABAAEAAIBRAA7").



- ✓ **Objetos de fecha y hora:** representan fechas y horas en el tiempo. Ejemplo: `ISODate("2021-10-01T14:30:00.000Z")`.
- ✓ **Objetos de expresión regular:** representan patrones de texto. Ejemplo: `/hola/i`.
- ✓ **Null:** es un **valor nulo** que representa la ausencia de un valor. Ejemplo: `null`.

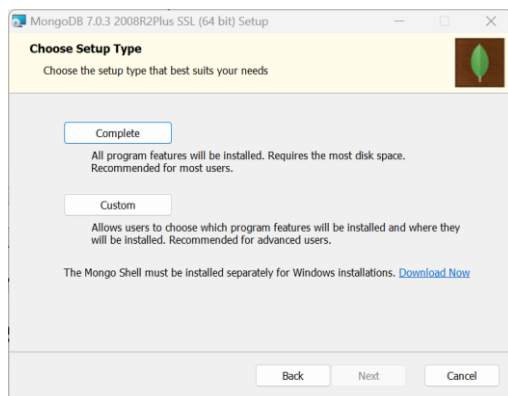
3.4 Instalación.

Los pasos a seguir para **instalar MongoDB Community Server en Windows** son:

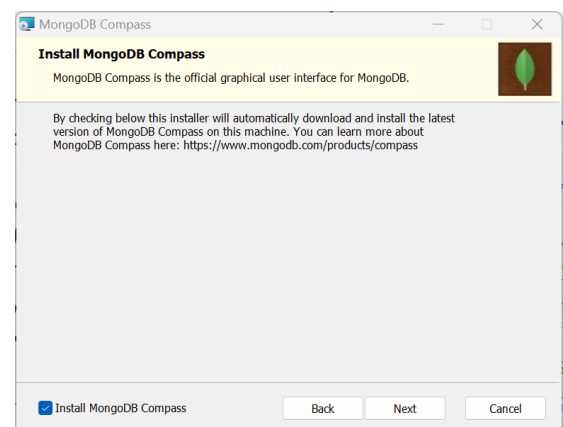
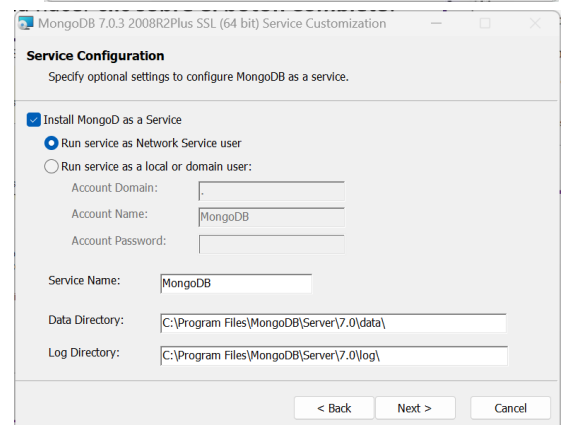
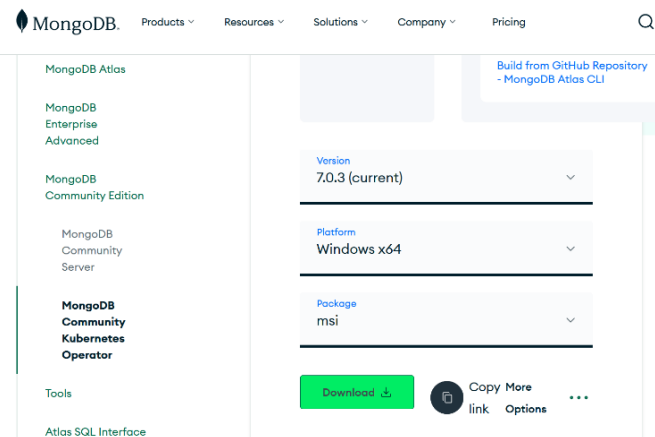
1. **Descargar MongoDB Community Server para Windows** desde el sitio web oficial de MongoDB: <https://www.mongodb.com/try/download/community>.
2. **Ejecutar el archivo de instalación** descargado y seguir las instrucciones del instalador.



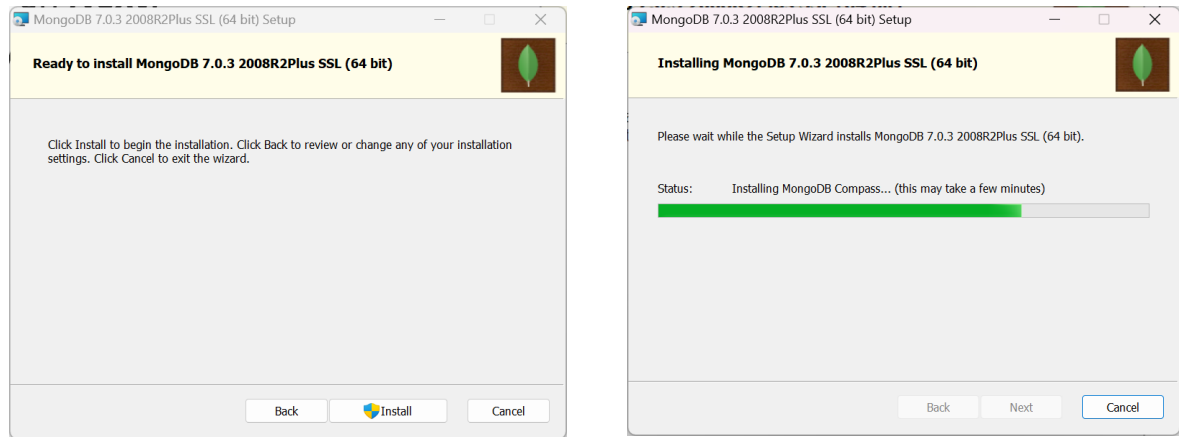
3. En la pantalla de bienvenida hacer clic en **Next**.
4. **Marcar la casilla de aceptación de los términos de la licencia** y hacer clic en **Next**.
5. En la siguiente ventana hacer **clic sobre el botón Complete**.



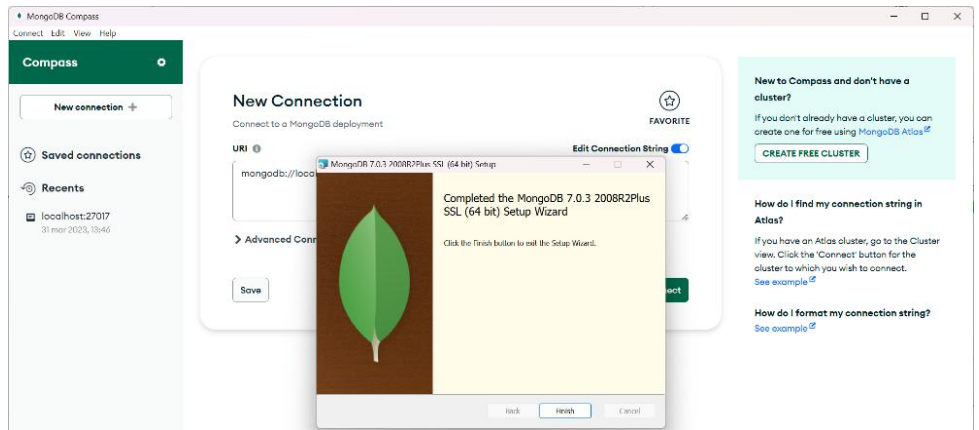
6. En la pantalla de "Configuración de inicio de servicio", **seleccionar la opción "Instalar MongoDB como servicio de Windows"** y especificar el nombre del servicio si se desea. Hacer clic en **Next**.
7. En la siguiente ventana dejar marcada la opción "Install MongoDB Compass" y hacer clic en **Next**.



8. Finalmente hacer clic en **Install** y esperar a que se complete el proceso de instalación. Permitir que la aplicación haga cambios en el sistema y esperar a que termine la instalación.



9. Una vez termine hacer clic en **Finish** en la ventana de instalación. Se abre en una ventana de fondo *MongoDB Compass*.



Además, al instalar *MongoDB* se instala **MongoDB Shell** (mongosh) que es una **interfaz de línea de comandos para interactuar con BD MongoDB**. Permite ejecutar diversos comandos y consultas contra **BD** y colecciones de *MongoDB*.

Instalar MongoDB Community con Docker: contenido del fichero `mongodb.yml`

```
# docker-compose -f mongodb.yml -p mongodb up -d
# mongosh -u root -p y usar root como clave
# version: '3.0'
services:
  mongodb:
    image: mongo
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: root
    ports:
      - 27017:27017
    networks:
      - mongo
# volumes:
# - mongodb_data:/data/db
networks:
  mongo:
    driver: bridge
#volumes:
# mongodb_data:
```

Descargar **MongoDB Shell** de forma independiente:

<https://www.mongodb.com/try/download/shell>

Ejecución del comando `mongod -version` desde dentro del contenedor para ver la versión de *MongoDB* instalada.

```
root@6dba980988cc:/# mongod -version
db version v7.0.4
Build Info: {
  "version": "7.0.4",
  "gitVersion": "38f3e37057a43d2e9f41a39142681a76062d582e",
  "opensslVersion": "OpenSSL 3.0.2 15 Mar 2022",
  "modules": [],
  "allocator": "tcmalloc",
  "environment": {
    "distmod": "ubuntu2204",
    "distarch": "x86_64",
    "target_arch": "x86_64"
  }
}
root@6dba980988cc:/#
```

El archivo `mongodb.yml` configura un contenedor de *MongoDB* con autenticación habilitada. Define el usuario administrador `root` con contraseña `root` y expone el puerto 27017 para conexiones externas. Para conectar al *SGBD MongoDB* haciendo uso de la *Shell*: `mongosh [--host <HOST_IP>] [--port <PORT>] -u <USERNAME> [-p]`.

Para iniciar este servicio, solo se necesita ejecutar el comando `docker-compose -f mongodb.yml -p mongodb up -d` en el directorio donde se encuentre el archivo `mongodb.yml` con el contenido anterior propuesto.

Por defecto, **MongoDB** utiliza el **puerto 27017** para las conexiones de red. Sin embargo, también es posible configurar *MongoDB* para que use un puerto diferente si es necesario. Además del puerto 27017, *MongoDB* también utiliza otros puertos para diferentes propósitos, como el puerto 27018 para la replicación interna y el puerto 27019 para la comunicación entre miembros de un conjunto de réplicas. Es importante tener en cuenta que el **puerto de MongoDB debe estar abierto en el firewall del servidor y en la red para permitir conexiones externas**.

Tutoriales sobre el proceso de **instalación** <https://docs.mongodb.com/manual/installation/>

3.5 Herramientas gráficas.

3.5.1 MongoDB Compass.

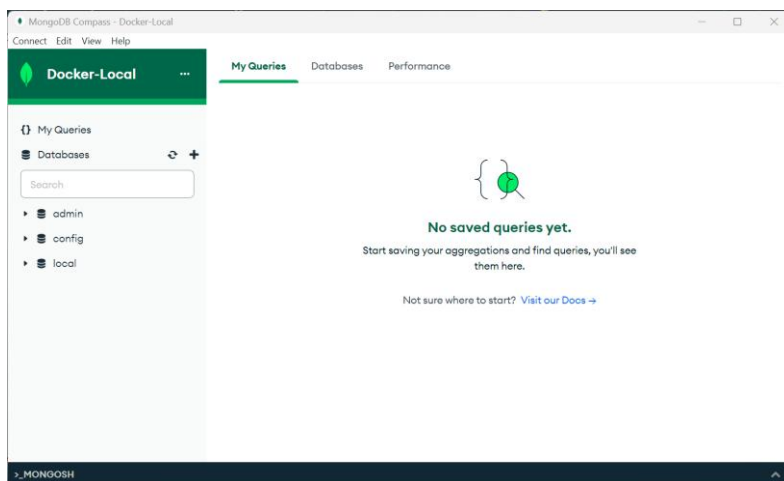
MongoDB Compass es una **potente GUI** para **consultar, agregar y analizar datos de MongoDB en un entorno visual.**

Compass es **de uso gratuito** y **se puede ejecutar en macOS, Windows y Linux.**

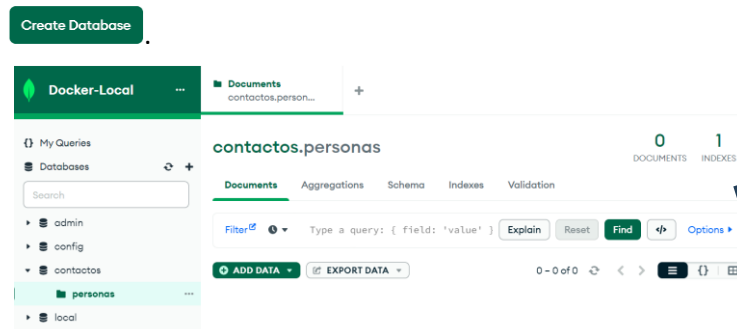
Se puede descargar de forma gratuita desde el sitio web oficial de *MongoDB* (en *Windows* se instala por defecto si se deja marcada la opción al realizar la instalación): <https://www.mongodb.com/try/download/compass>.

Para **conectar al servidor MongoDB** hacer **click en Connect**. Si se ha establecido alguna clave para el usuario root o bien se quiere acceder con algún otro usuario/clave hacer clic en **Advanced Connection Options**, seleccionar la pestaña **Authentication** y como método de autenticación seleccionar *Username/Password*. Si además de conectar, se desea guardar la conexión hacer clic en **Save & Connect**.

Una vez conectado se mostrará una ventana como la siguiente:

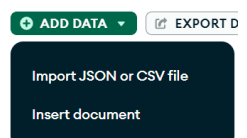
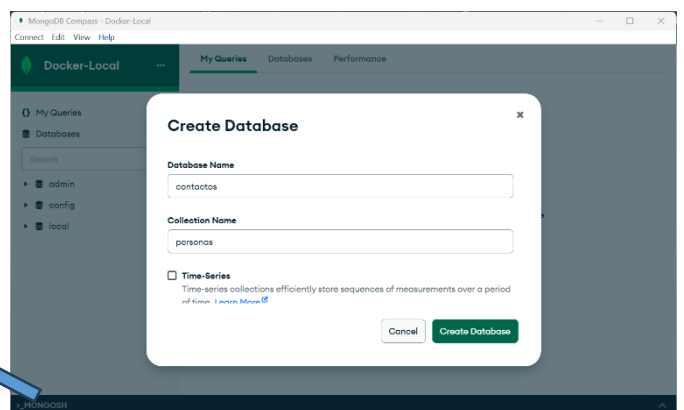
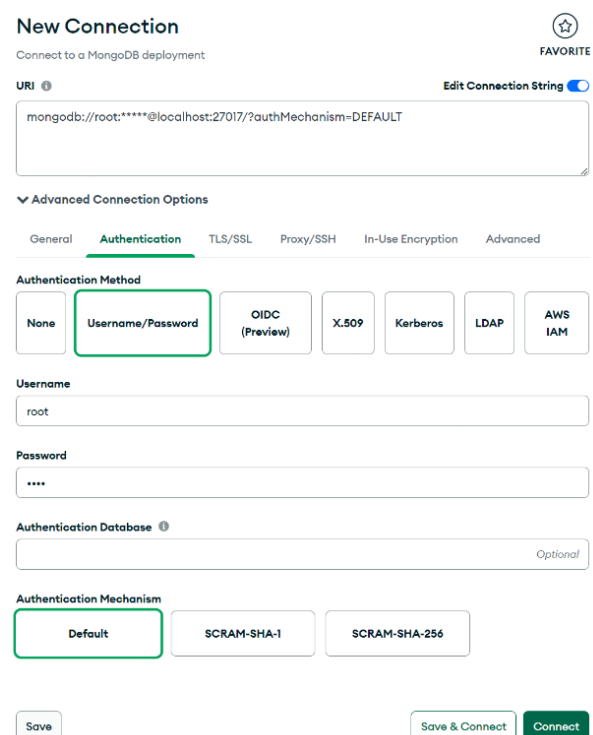
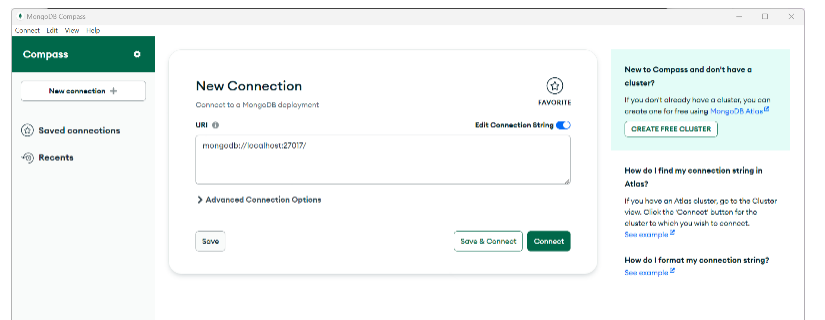


Para crear una nueva *BD* hacer clic en el botón **+** que aparece en la parte derecha de **Databases**. En la ventana que se muestra introducir el nombre de la *BD* a crear y el nombre de la primera colección. Finalmente hacer clic en el botón **Create Database**.



Como puede comprobarse, se dispone de varios botones que permiten agregar datos (manualmente o bien desde *JSON/CSV* de forma masiva), importar/exportar datos, filtros, buscar, ...

Nota: una convención utilizada en *MongoDB* es **nombrar las colecciones en plural.**



MongoDB Compass tiene incorporado **MongoDB Shell** (**mongosh** - interfaz de línea de comandos para interactuar con **BD MongoDB**), simplemente una vez conectado a un servidor **MongoDB** se debe desplegar en la parte inferior de la ventana.

MongoDB Shell proporciona una interfaz basada en **JavaScript**, lo que significa que se puede escribir y ejecutar código **JavaScript** directamente en la consola (2+2, print("Hola mundo!"), x=200, x, new Date(), Math.sin(90), etc.). Esto facilita la realización de consultas y manipulaciones de datos complejas.

```
>_MONGOSH
> function factorial(n) {if (n <= 1) return 1; return n * factorial(n-1)}
< [Function: factorial]
> factorial(5)
< 120
test>
```

Más información sobre el uso de MongoDB Compass <https://www.mongodb.com/docs/compass/current/>

3.5.2 Studio 3T.

Studio 3T es una **herramienta que facilita el desarrollo y la administración de BD MongoDB**, ofreciendo una interfaz gráfica fácil de usar y diversas características para mejorar la productividad de los desarrolladores y administradores de **BD**.



Características principales:

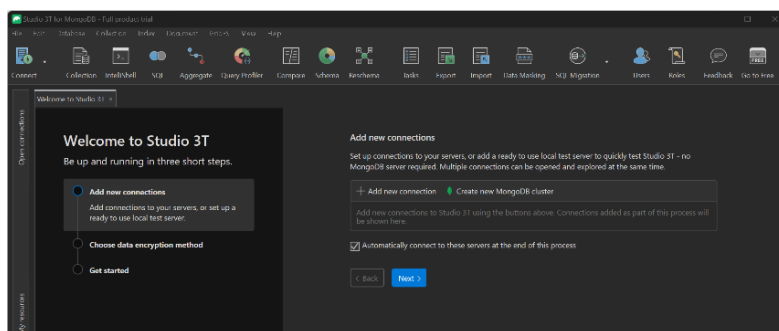
- ✓ **Interfaz gráfica de usuario (GUI):** proporciona una interfaz visual intuitiva que facilita la interacción con **BD MongoDB** sin necesidad de comandos de consola.
- ✓ **Editor de consultas:** permite escribir y ejecutar consultas **MongoDB** de manera eficiente.
- ✓ **Explorador de datos:** facilita la visualización y la manipulación de datos almacenados en las colecciones de **MongoDB**.
- ✓ **Importación y exportación de datos:** ofrece herramientas para importar y exportar datos de y hacia **MongoDB**.
- ✓ **Generación de consultas visual:** ayuda a construir consultas complejas de manera visual.
- ✓ **Soporte para agregaciones:** permite la construcción y ejecución de pipelines de agregación, que son poderosas herramientas de análisis de datos en **MongoDB**.

Studio 3T ofrece diferentes versiones, se dispone de una **versión gratuita con funcionalidades básicas** y versiones comerciales con características avanzadas.

Es **compatible con diversas versiones de MongoDB**, lo que permite a los usuarios trabajar con diferentes versiones de la **BD**.

Puede integrarse con **MongoDB Compass** para proporcionar funcionalidades adicionales.

Studio 3T suele recibir **actualizaciones periódicas** para mantenerse al día con las nuevas versiones de **MongoDB** y agregar características mejoradas. También **cuenta con una comunidad activa** que comparte experiencias y brinda soporte.



Descarga y más información en <https://studio3t.com/es/>

MongoDB Visual Studio Code Plugin: <https://code.visualstudio.com/docs/azure/mongodb>

3.6 MongoDB Atlas.

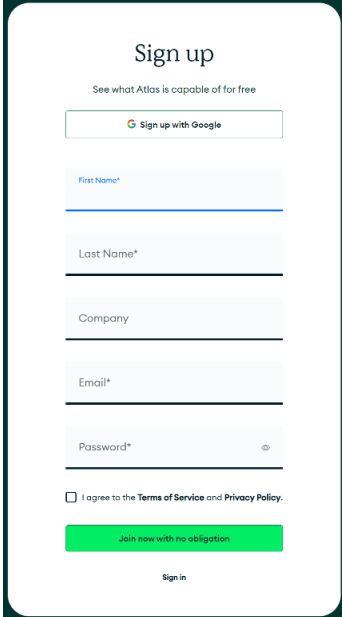
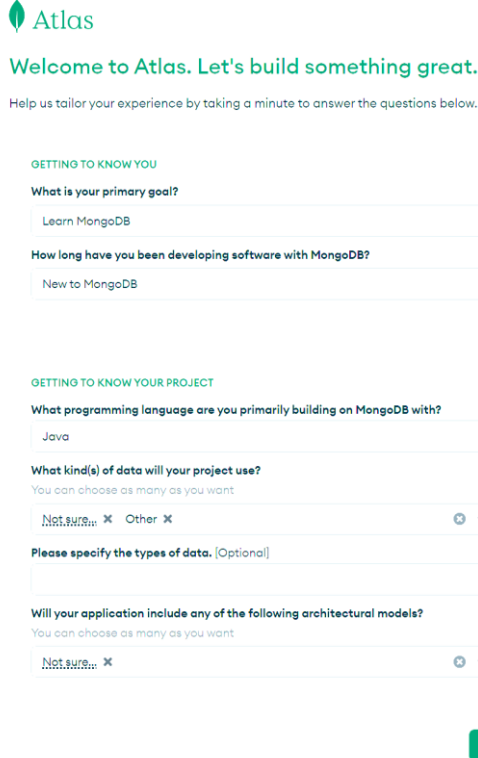
Es un **servicio de BD en la nube totalmente administrado por MongoDB**. Es una plataforma de **BD** como servicio (**DBaaS**) que ofrece una manera fácil y escalable de implementar, administrar y escalar **BDs MongoDB**.

Se ejecuta en **múltiples proveedores de la nube** (**AWS**, **Google Cloud Platform** y **Microsoft Azure**), lo que permite a los usuarios elegir la infraestructura que mejor se adapte a sus necesidades.

MongoDB Atlas proporciona una forma sencilla de alojar y gestionar datos en la nube.

Para **comenzar a utilizar MongoDB Atlas**, seguir los siguientes **pasos**:

1. **Acceder al sitio web de Atlas** (<https://www.mongodb.com/atlas>).
2. **Crear una cuenta Atlas**. Registrarse para obtener una cuenta *Atlas* utilizando una cuenta de *Google* o una dirección de correo electrónico.
 - a. Hacer clic en el botón  para crear una cuenta gratuita.
 - b. Completar el formulario de registro y seguir las instrucciones para crear una nueva organización y un nuevo proyecto. Otra opción disponible es hacer uso de una cuenta de *Google*.
 - c. Verificar la dirección de e-mail haciendo clic sobre el enlace que llega al e-mail indicado.
 - d. Una vez verificada la cuenta se redirigirá a una ventana de bienvenida en la que se deberá responder a varias preguntas, tras lo cual finalmente se hará clic en el botón *Finish*.

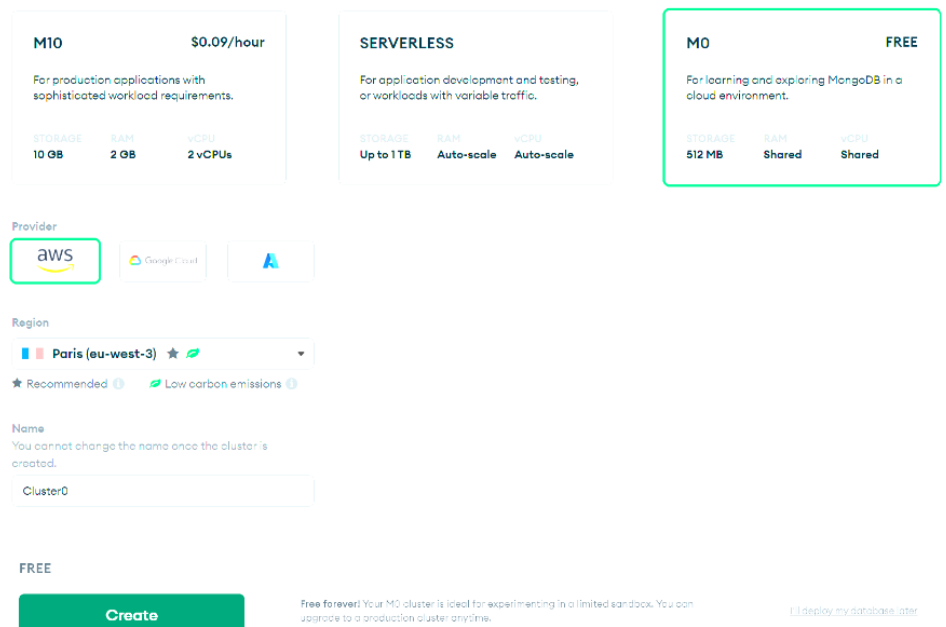



3. Crear un clúster gratuito.

Seleccionar la configuración *M0* (que es gratuita), seleccionar un proveedor del servicio (por ejemplo, *AWS*), seleccionar una de las regiones recomendadas (la más próxima para reducir en lo posible las latencias) y asignar un nombre al clúster. Finalmente hacer clic en el botón *Create*.

MongoDB Deploy your database

Use a template below or set up [advanced configuration options](#). You can also edit these configuration options once the cluster is created.



4. **Crear un usuario de BD para el clúster.** En la siguiente página indicar como se va a autenticar la conexión. Escoger la opción *Username and Password* y crear un usuario.

✓ How would you like to authenticate your connection?

Your first user will have permission to read and write any data in your project.

Username and Password

Certificate

📘 We autogenerated a username and password for your first database user in this project using your MongoDB Cloud registration information. ✕

Create a database user using a username and password. Users will be given the *read and write to any database privilege* by default. You can update these permissions and/or create additional users later. Ensure these credentials are different to your MongoDB Cloud username and password. You can manage existing users via the [Database Access Page](#).

Username

Enter username

Password

Enter password

🔑 Autogenerate Secure Password

📋 Copy

Create User

Username	Authentication Type
██████████	Password

✎ EDIT 🗑 REMOVE

5. **Agregar la dirección IP de la conexión a la lista de direcciones IP confiables.** A un clúster solo se puede acceder desde una dirección IP confiable. Seleccionar *My Local Environment* y añadir la dirección IP de la conexión.
Nota: si se establece el valor 0.0.0.0/0 se permite la conexión desde cualquier dirección IP.

✓ Where would you like to connect from?

Enable access for any network(s) that need to read and write data to your cluster.

My Local Environment

Use this to add network IP addresses to the IP Access List. This can be modified at any time.

Cloud Environment

Use this to configure network access between Atlas and your cloud or on-premise environment. Specifically, set up IP Access Lists, Network Peering, and Private Endpoints.

📘 We added your current IP address. You can connect to your cluster locally from this device. ✕

Add entries to your IP Access List

Only on IP address you add to your Access List will be able to connect to your project's clusters. You can manage existing IP entries via the [Network Access Page](#).

IP Address	Description
Enter IP Address	Enter description

Add My Current IP Address

Add Entry

IP Address List	Description
██████████	

🎉 Congratulations on setting up access rules!

You will now be able to connect to your deployments. You can continue to add and update access rules in [Database Access](#) and [Network Access](#).

☒ Hide Quickstart guide in the navigation. You can visit [Project Settings](#) to access it in the future.

Go to Overview

6. Finalmente hacer clic en **Finish and Close**.
7. En la ventana emergente que se muestra hacer clic en el botón **Go to Overview**.
8. Se accederá a una nueva página en la que se muestra información sobre el clúster de BD que se acaba de desplegar en *MongoDB Atlas*.

Overview

Database Deployments

Cluster0

CONNECT EDIT CONFIGURATION

FREE SHARED

Add Data Load Sample Data Data Modeling Templates

+ Add Tag

Desde la propia página web se podrán crear BD y trabajar directamente con las colecciones (**Database** → **Browse Collections**).

JUAN LUIS'S ORG - 2023-12-30 > PROJECT 0 > DATABASES

Cluster0

VERSION: 6.0.12 REGION: AWS Paris (eu-west-3)

Overview Real Time Metrics Collections Atlas Search Profiler Performance Advisor Online Archive Cmd Line Tools

DATABASES: 1 COLLECTIONS: 1

+ Create Database

Search Namespaces

conferencias

ponentes

conferencias.ponentes

STORAGE SIZE: 4KB LOGICAL DATA SIZE: 0B TOTAL DOCUMENTS: 0 INDEXES TOTAL SIZE: 4KB

Find Indexes Schema Anti-Patterns Aggregation Search Indexes






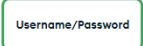


INSERT DOCUMENT

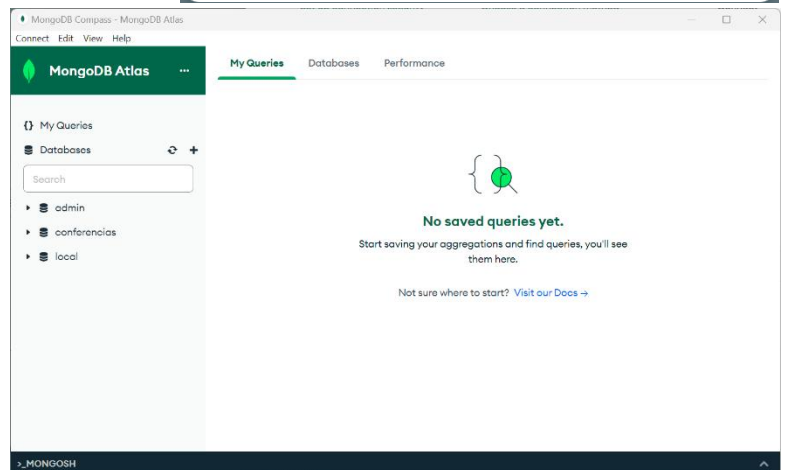
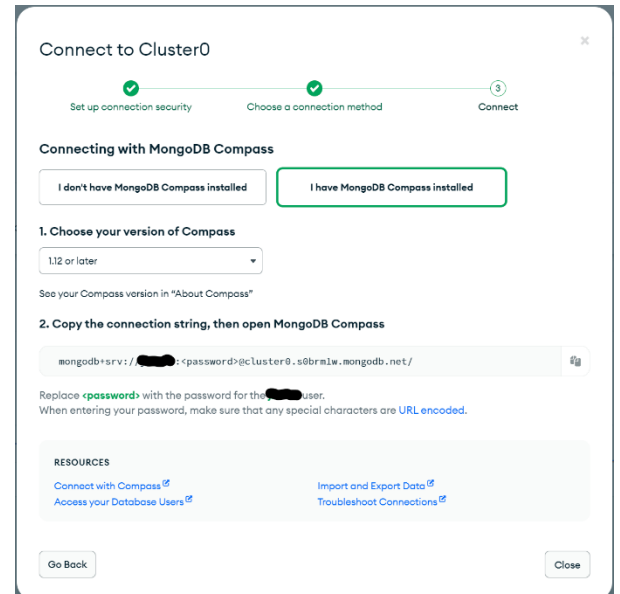
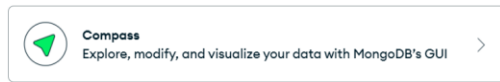
Filter Type a query: { field: 'value' }

Reset Apply Options

QUERY RESULTS: 0

Otra opción es **conectarse al clúster haciendo uso de MongoDB Compass**:

- Para ello hacer **clic en el botón**  (se puede acceder desde *Overview* o *Database*) para obtener la cadena de conexión a la *BD*. En la ventana emergente que se abre **seleccionar**
- En la siguiente ventana **seleccionar** . Se mostrará la cadena de conexión que se debe utilizar para conectar haciendo uso de *MongoDB Compass*, **copiar la cadena de conexión** haciendo clic en .
- Para conectar al clúster **desde MongoDB Compass crear una nueva conexión y pegar en el campo URI la cadena de conexión** anteriormente copiada.
- Desplegar** , **seleccionar la pestaña**  y como **método de autenticación** **seleccionar** .
- Indicar el usuario y clave** utilizado para crear el clúster.
- En el mecanismo de autenticación dejar el que se muestra por defecto.
- Finalmente hacer **clic en el botón**  o  si se desea guardar la conexión.
- Se establecerá una conexión con el clúster de *MongoDB Atlas* y se podrá trabajar con el tal como se trabaja con la instalación local de *MongoDB*.



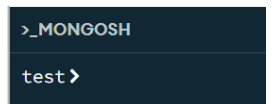
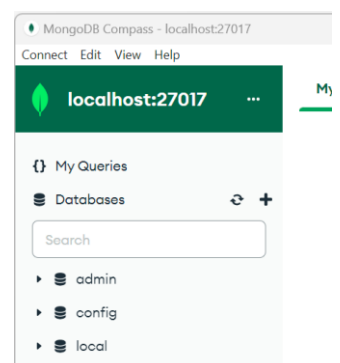
3.7 BD del sistema.

Existen varias *BD* de administración que **se crean automáticamente cuando se inicia una instancia de MongoDB** (sirven para el funcionamiento interno de *MongoDB*). Estas *BD* son:

- ✓ **admin**: se utiliza para **administrar la instancia de MongoDB**. En esta *BD* es donde se pueden **crear y administrar usuarios y roles**, así como también se pueden ver las estadísticas de la instancia y **configurar opciones de seguridad**.
- ✓ **config**: se utiliza para **almacenar la configuración de un conjunto de réplicas de MongoDB**. En esta *BD* es donde se guarda la información sobre la topología del conjunto de réplicas, como los miembros del conjunto y sus roles.
- ✓ **local**: se utiliza para **almacenar datos locales para una instancia de MongoDB**. En esta *BD* es donde se almacena la información sobre las operaciones de replicación y los bloques de datos utilizados por el motor de almacenamiento de *MongoDB*.

Es importante tener en cuenta que **las BD admin y config solo deben ser utilizadas por usuarios con los permisos adecuados y para administrar la instancia de MongoDB**. La *BD local* no debe ser modificada por los usuarios, ya que contiene información crítica para el correcto funcionamiento de la instancia de *MongoDB*.

Además de estas tres *BD* por defecto, **los usuarios pueden crear otras BD en MongoDB**. Cuando se conecta al servidor de *MongoDB* la *BD* por defecto que se tiene es *test*.



3.8 Comandos básicos (MongoDB Shell).

MongoDB Shell es la forma más rápida de conectarse y trabajar con *MongoDB*. Permite consultar datos,

configurar ajustes y ejecutar otras acciones fácilmente. Es una interfaz de línea de comandos moderna y extensible, con resaltado de sintaxis, autocompletado inteligente, ayuda contextual y mensajes de error. Se puede descargar desde <https://www.mongodb.com/try/download/shell>.

Para **conectar con un servidor de MongoDB** haciendo uso de *MongoDB Shell* ejecutar:

```
mongosh "mongodb+[srv]://<servidor_mongodb>/[<bd_inicial>]"
--apiVersion 1 --username <username>
```

```
mongosh [--host <servidor_mongodb> --port <puerto>] -u <username>
```

, donde:

- +srv: opcional, se utiliza para conectar con clúster de *MongoDB Atlas*, en caso de conectar con un servidor local no es necesario.
- <servidor_mongodb>: dirección IP o nombre asignado al servidor/clúster de *MongoDB*.
- <bd_inicial>: BD que se selecciona al iniciar sesión. Es opcional y puede omitirse.
- <username>: nombre de usuario con el que conectar al servidor/clúster de *MongoDB*.

Para obtener ayuda: `mongosh -h`

Ejemplo: conexión a un clúster de *MongoDB Atlas*.

```

C:\Users\jl\Downloads\mongosh-2.1.1-win32-x64\bin>mongosh "mongodb+srv://cluster0.s0brmlw.mongodb.net/conferencias" --apiVersion 1 --username jlmoru
Enter password: *****
Current Mongosh Log ID: 65901782e5536af73a3a1724
Connecting to:      mongodb+srv://<credentials>@cluster0.s0brmlw.mongodb.net/conferencias?appName=mongosh+2.1.1
Using MongoDB:      6.0.12 (API Version 1)
Using Mongosh:      2.1.1

For mongosh info see: https://docs.mongodb.com/mongosh-shell/

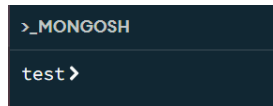
Atlas atlas-6zvcso-shard-0 [primary] conferencias>

```

Ejemplo: conexión a un servidor local de *MongoDB* desplegado con *Docker*.

```
mongosh "mongodb://localhost/" --apiVersion 1 --username root
mongosh --host localhost --port 27017 -u root
```

Otra forma de hacer uso de *MongoDB Shell* es utilizar *MongoDB Compass*, en la parte inferior izquierda de la ventana hacer clic sobre **>_MONGOSH**.



Comandos básicos (casi todos los comandos se escriben en minúsculas):

- **db**: obtener el **nombre de la BD con la que se está trabajando** (actual). Otra opción es utilizar `db.getName()`.
- **show dbs** (show databases): obtener un **listado de las BD** creadas en el servidor/clúster de *MongoDB*.
- **use <nombre_bd>**: **seleccionar una BD** para trabajar con ella. Si no existe la crea, la nueva BD creada no se mostrará hasta que no contenga al menos un documento.
- **db.dropDatabase()**: **eliminar la BD actual** (esta operación es irreversible).
- **cls**: limpiar la *shell* de *MongoDB*.
- **help**: lista de **comandos**.
- **db.help()**: ayuda **métodos BD**.
- **db.<nombre_colección>.help()**: ayuda **métodos colecciones**.
- **// y /* ... */**: **comentarios**.
- **show collections** (show tables): **mostrar las colecciones** que hay en la BD seleccionada.
- **db.<nombre_colección>.countDocuments()**: **número de documentos** que contiene la colección.
- **db.<nombre_colección>.find()**: **ver toda la información que hay dentro de una colección**.
- **db.<nombre_colección>.find().pretty()**: dar **formato JSON** al resultado.
- **db.<nombre_colección>.find({filtros}).limit(n)**: **limitar número de documentos que devuelve la consulta**. filtros es un objeto que define los criterios de búsqueda de documentos en una colección.
- **db.<nombre_colección>.find({filtros}).count()**: **número de documentos que devuelve la consulta**. Otra opción: `db.<nombre_colección>.countDocuments({filtros})`
- **db.<nombre_colección>.find({_id: ObjectId(<id_objeto>)})**: **ver la información de un documento específico dentro de una colección**.

```

test> show dbs
admin      100.00 KiB
config     72.00 KiB
contactos  8.00 KiB
local      72.00 KiB
test> use contactos
switched to db contactos
contactos> use empresa
switched to db empresa
empresa> show dbs
admin      100.00 KiB
config     96.00 KiB
contactos  8.00 KiB
local      72.00 KiB

```

- `db.<nombre_colección>.find(<{clave_búsqueda>: <coincidencia_por_la_que_buscar>})`: buscar documentos por alguna coincidencia exacta dentro de una colección.
- `db.<nombre_colección>.insertOne(<objeto_json>)`: insertar un documento en la colección. Si se intenta insertar un documento en una colección que no existe en la BD, *MongoDB* lo detecta y la crea antes de proceder a insertar el documento.

```
contactos> db.personas.find({"nombre": "Juan"})
[
  {
    _id: ObjectId('65904f70023ae21b9400032d'),
    nombre: 'Juan',
    apellidos: 'Moreno',
    edad: 30
  }
]
```

```
empresa> use contactos
switched to db contactos
contactos> show collections
personas
contactos> db.personas.find()

contactos> db.personas.insertOne({ "nombre" : "Juan" , "apellidos" : "Moreno", "edad" : 30 })
{
  acknowledged: true,
  insertedId: ObjectId('65904f70023ae21b9400032d')
}
contactos> db.personas.find({_id: ObjectId('65904f70023ae21b9400032d')})
[
  {
    _id: ObjectId('65904f70023ae21b9400032d'),
    nombre: 'Juan',
    apellidos: 'Moreno',
    edad: 30
  }
]
```

- `db.<nombre_colección>.insertMany([<objeto_json1>, <objeto_json2>, ...])`: insertar más de un documento a la vez en la colección. Entre los corchetes, que indican la existencia de un array, se indican los distintos objetos que representan los documentos a insertar.
- `Date()`: devuelve la fecha y hora del sistema.
- `exit / exit() / .exit / quit / quit()`: salir de *MongoDB Shell*.

3.9 Colecciones, documentos y BSON.

JSON, o notación de objetos *JavaScript*, es un estándar muy popular para el intercambio de datos en la web, y es en el que se basa **BSON** (**B**inary **J**SON).

Los **objetos JSON** son contenedores asociativos, en los que a una clave de cadena se le asigna un valor (que puede ser un número, una cadena, un valor booleano, una matriz, un valor vacío, nulo o incluso otro objeto). Casi cualquier lenguaje de programación tiene una implementación para esta estructura de datos abstracta: objetos en *JavaScript*, diccionarios en *Python*, tablas hash en *Java* y *C#*, matrices asociativas en *C++*, etc. Los objetos *JSON* son fáciles de entender para los humanos y para que las máquinas los analicen y generen (ver imagen).

	JSON	BSON
Codificación	Cadena UTF-8	Binario
Soporte de datos	Cadena, booleano, número, matriz, objeto, nulo	Cadena, booleano, número (entero, flotante, largo, decimal128...), matriz, nulo, fecha, BinData
Legibilidad	Humano y máquina	Sólo máquina

Hoy en día, **JSON** se utiliza en muchos ámbitos diferentes:

- APIs.
- Archivos de configuración.
- Registro de mensajes.
- Almacenamiento de BDs.

MongoDB fue diseñado desde su inicio para ser una BD enfocada en brindar una excelente experiencia de desarrollo. La generalidad de *JSON* lo convirtió en la opción obvia para representar estructuras de datos en el modelo de datos de documentos de *MongoDB*.

```
{
  "_id": 1,
  "name": { "first": "John", "last": "Backus" },
  "contribs": [ "Fortran", "ALGOL", "Backus-Naur Form", "FP" ],
  "awards": [
    {
      "award": "W.W. McDowell Award",
      "year": 1967,
      "by": "IEEE Computer Society"
    }, {
      "award": "Draper Prize",
      "year": 1993,
      "by": "National Academy of Engineering"
    }
  ]
}
```

Sin embargo, existen varios **problemas que hacen que JSON no sea ideal** para su uso dentro de una BD:

- *JSON* solo admite una **cantidad limitada de tipos de datos** básicos. En particular, *JSON* carece de soporte para fechas y datos binarios.
- Los **objetos y propiedades JSON** no tienen una longitud fija, lo que hace que el recorrido sea más lento.

Para hacer que *MongoDB* usara *JSON*, pero aun así fuese de alto rendimiento y propósito general, se inventó **BSON** para cerrar la brecha. **BSON** es una **representación binaria para almacenar datos en formato JSON, optimizada para velocidad, espacio y eficiencia**.

La estructura binaria de **BSON** **codifica información de tipo y longitud**, lo que permite recorrerla mucho más rápidamente en comparación con *JSON*.

BSON agrega algunos tipos de datos no nativos de JSON, como fechas y datos binarios, sin los cuales a *MongoDB* le habría faltado un soporte valioso.

En la imagen se pueden ver dos ejemplos de objetos *JSON* y sus representaciones en *BSON*.

Especificación de *BSON* en <https://bsonspec.org/>

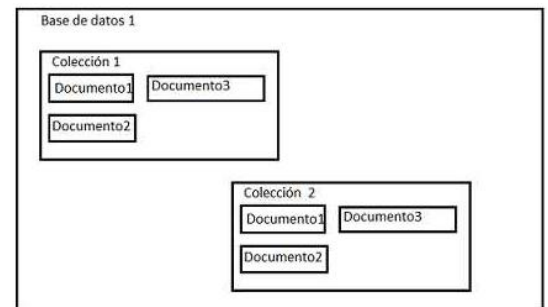
MongoDB almacena los datos en formato BSON, tanto internamente como en la transmisión por red. Sin embargo, esto no impide que se pueda considerar como una *BD* orientada a *JSON*. Cualquier dato representable en *JSON* se puede almacenar de forma nativa en *MongoDB* y recuperarse fácilmente en el mismo formato.

```
{ "hello": "world" } →
\x16\x00\x00\x00      // total document size
\x02                   // 0x02 = type String
hello\x00              // field name
\x06\x00\x00\x00world\x00 // field value
\x00                   // 0x00 = type E00 ('end of object')
```

```
{ "BSON": [ "awesome", 5.05, 1986 ] } →
\x31\x00\x00\x00      // total document size
\x04BSON\x00          // field name
\x26\x00\x00\x00      // field value
\x02\x30\x00\x00\x08\x00\x00\x00awesome\x00
\x01\x31\x00\x00\x33\x33\x33\x33\x33\x33\x14\x40
\x10\x32\x00\x00\x07\x00\x00
\x00
\x00
```

MongoDB utiliza *BSON* internamente para guardar la información y su tráfico de red, pero el usuario utiliza *JSON* para interactuar con él.

MongoDB es una *BD* documental donde el elemento esencial es el **documento**. Estos documentos **suelen agruparse en colecciones** de estructura similar. En *MongoDB*, una *BD* está compuesta por un **conjunto de colecciones** (ver imagen).



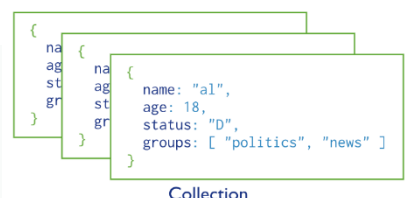
MongoDB almacena registros de datos como documentos (específicamente documentos *BSON*) que se reúnen en colecciones. Una *BD* almacena una o más colecciones de documentos.

MongoDB almacena documentos en colecciones. Las colecciones son análogas a las tablas de las *BD* relacionales.

Los **documentos MongoDB** se componen de pares de **campo-valor** y tienen la siguiente estructura:

El **valor de un campo puede ser cualquiera de los tipos de datos BSON**, incluidos otros documentos, matrices y matrices de documentos. Por ejemplo, el siguiente documento contiene valores de distintos tipos:

```
{
  field1: value1,
  field2: value2,
  field3: value3,
  ...
  fieldN: valueN
}
```



En *MongoDB* las comillas dobles en los nombres de los campos son opcionales si el nombre es simple (sin espacios o caracteres especiales). En *JSON* es obligatorio usar comillas dobles.

```
var mydoc = {
  _id: ObjectId("5099803df3f4948bd2f98391"),
  name: { first: "Alan", last: "Turing" },
  birth: new Date('Jun 23, 1912'),
  death: new Date('Jun 07, 1954'),
  contribs: [ "Turing machine", "Turing test", "Turingery" ],
  views : NumberLong(1250000)
}
```

El **tamaño máximo de documento BSON en MongoDB es de 16 MBytes** (limita la cantidad de datos a almacenar).

Los **nombres de los campos son cadenas**. Los documentos tienen las siguientes **restricciones en los nombres de los campos**:

- El **nombre del campo _id está reservado para su uso como clave principal**. Su valor debe ser único en la colección, inmutable (no puede cambiarse una vez asignado) y puede ser de cualquier tipo que no sea un arreglo. Si **_id** contiene subcampos, los nombres de los subcampos no pueden comenzar con un símbolo \$.
- **No se puede usar null como nombre de campo**.
- El servidor permite el almacenamiento de nombres de campos que contienen puntos (.) y signos de dólar (\$).

3.10 Operaciones con colecciones.

MongoDB es una *BD* no relacional que utiliza colecciones y documentos para almacenar y organizar la información. Las **operaciones** que se pueden realizar en las colecciones son:

- ✓ **Mostrar colecciones:** se pueden mostrar las colecciones existentes en la *BD* seleccionada con el comando `show collections` (`show tables`).
- ✓ **Creación de colecciones:** se pueden crear nuevas colecciones con el **método** `createCollection()` que se aplica sobre el objeto `db`. Su sintaxis es:

```
db.createCollection("<nombre_colección>" [, <opciones>])
```

, donde:

- **<nombre_colección>:** es el nombre de la colección que se va a crear. Es una cadena de caracteres y es obligatorio.
- **<opciones>:** es un **objeto que especifica las opciones adicionales para la creación de la colección**. Es opcional. Algunas opciones que se pueden incluir son:
 - **capped:** especifica si la colección es una colección **limitada en tamaño**. El valor debe ser `true` o `false`. Por defecto, las colecciones no son limitadas y pueden crecer indefinidamente.
 - **size:** especifica el **tamaño máximo de la colección si es una colección limitada**. Es un número en bytes.
 - **max:** especifica el **número máximo de documentos que puede tener la colección si es una colección limitada**. Es un número entero.

Ejemplo: crear una colección llamada `logs` que tenga un tamaño máximo de 1.000.000 bytes y un máximo de 1.000 documentos.

```
db.createCollection("logs", {capped: true, size: 1000000, max: 1000})
```

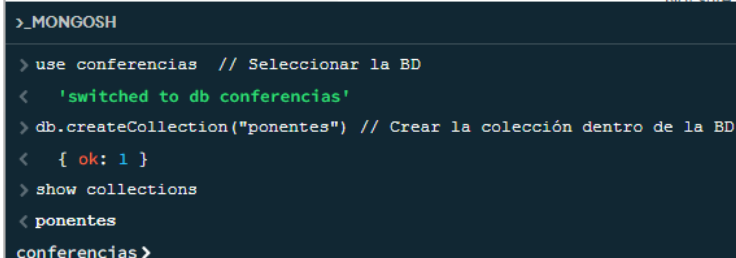
Ejemplo: crear una colección llamada `ponentes` en la *BD* `conferencias`.

```
use conferencias // Seleccionar la BD "conferencias"
```

```
db.createCollection("ponentes") // Crear la colección dentro de la BD seleccionada
```

Para ver la lista de colecciones existentes en la *BD* actual, incluida la que se acaba de crear, utilizar el comando:

```
show collections
```



```
>_MONGOSH
> use conferencias // Seleccionar la BD
< 'switched to db conferencias'
> db.createCollection("ponentes") // Crear la colección dentro de la BD
< { ok: 1 }
> show collections
< ponentes
conferencias>
```

- ✓ **Eliminar colecciones:** se utiliza el **método** `drop()` sobre la colección correspondiente. Su sintaxis es:

```
db.<nombre_colección>.drop()
```

, donde:

- **<nombre_colección>:** es el **nombre de la colección que se desea eliminar**.

Ejemplo: eliminar la colección `usuarios` y todos sus documentos.

```
db.usuarios.drop()
```

- ✓ **Renombrar colecciones:** se utiliza el **método** `renameCollection()` sobre la colección correspondiente. La sintaxis básica de este método es la siguiente:

```
db.<nombre_colección>.renameCollection("<nuevo_nombre_colección>" [, <drop_target>])
```

, donde:

- **<nuevo_nombre_colección>:** es el nuevo **nombre de la colección a la que se va a renombrar** la colección existente.
- **<drop_target>:** es un **valor booleano que indica si se debe eliminar la colección de destino si ya existe una colección con el nombre de la nueva colección**. Si se establece en `true`, *MongoDB* elimina la colección de destino antes de renombrar la colección de origen. Si se establece en `false` (o si no se especifica), se genera un error si la colección de destino ya existe. Es una opción opcional.

Ejemplos: `db.ponentes.renameCollection("ponentes2")`

```
db.ponentes2.renameCollection("ponentes", true)
```

3.11 CRUD de documentos.

3.11.1 Inserción.

Las **operaciones de creación/inserción añaden nuevos documentos a una colección**. Si la colección no existe, las operaciones de inserción la crearán.

Cuando se insertan o modifican caracteres con tildes en *MongoDB*, pueden surgir problemas relacionados con la **codificación de caracteres** si el entorno o los datos no están configurados para usar **UTF-8**, que es el **estándar que MongoDB utiliza para almacenar cadenas**.

Si se interactúa con *MongoDB* a través de una terminal, asegurarse de configurarla para que utilice UTF-8.

MongoDB proporciona los siguientes **métodos** para insertar documentos en una colección:

- **db.<nombre_colección>.insertOne(<documento>)**: inserta **un documento** en la colección dada.
- **db.<nombre_colección>.insertMany([<documento1>, <documento2>, ..., <documentoN>])**: inserta **varios documentos** en la colección dada.

En *MongoDB*, las operaciones de inserción **tienen como objetivo una única colección**. Todas las operaciones de escritura en *MongoDB* son **atómicas a nivel de un solo documento**.

Para las **fechas** se puede utilizar:

- **new ISODate("yyyy-mm-dd[Thh:mm:ssZ]")**: la parte de la **hora es opcional** y se debe **poner la T de tiempo para indicarla y la Z final** que forma parte del formato. Este es un formato universal y estándar.
- **new Date(yyyy, mm, dd[, hh, mm, ss])**: el mes de **enero corresponde a 0** y el de **diciembre a 11**, por lo que si se pone 2 corresponde a marzo y no a febrero.

Nota: los documentos que se insertan no es necesario que tengan los mismos campos.

```
contactos> db.personas.insertOne({nombre: "Pepe", "fecha_nacimiento": new ISODate("1970-01-01T17:30:45Z")})
{
  acknowledged: true,
  insertedId: ObjectId('659069f3023ae21b9400032e')
}
contactos> db.personas.insertOne({nombre: "Carlos", "fecha_nacimiento": new Date(1985, 02, 15, 14, 43, 35)})
{
  acknowledged: true,
  insertedId: ObjectId('65906a2e023ae21b9400032f')
}
```

Si se muestran todos los documentos de la colección, se obtendrá el resultado que se muestra en la imagen.

Se puede **crear una colección vacía** haciendo uso de **db.<nombre_colección>.insertOne({})**.

Quando se inserta un documento y no se proporciona un valor para el campo **_id**, *MongoDB* automáticamente asigna un valor único de 12 bytes a este campo. El campo **_id** es un campo especial en *MongoDB* y se utiliza como **clave principal en la colección**.

Para insertar varios documentos, los objetos deben pasarse dentro de un **arreglo** (entre corchetes []). Ejemplo:

```
contactos> db.personas.insertMany([{"nombre": "Jorge", "fecha_nacimiento": new ISODate("1995-01-10T17:35:15Z")},
  { nombre: "Marina", "fecha_nacimiento": new ISODate("2005-07-19T11:20:15Z")}])
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId('65906eb0023ae21b94000332'),
    '1': ObjectId('65906eb0023ae21b94000333')
  }
}
```

Nota: si se intenta insertar un documento con un valor de **_id** que ya exista en la colección, se producirá un error por duplicado y la operación de inserción fallará.

3.11.2 Actualización.

Las **operaciones de actualización modifican documentos existentes en una colección**. *MongoDB* proporciona los siguientes **métodos** para actualizar documentos de una colección:

- **db.<nombre_colección>.updateOne(<filtro>, <cambios> [, <opciones>])**: modifica un solo documento. El **primer documento que cumpla con el criterio de selección** especificado en <filtro>.
- **db.<nombre_colección>.updateMany(<filtro>, <cambios> [, <opciones>])**: modifica varios documentos. **Todos los documentos que coinciden con el criterio de selección** especificado en <filtro>.

- `db.<nombre_colección>.replaceOne({_id: ObjectId("id")}, <objeto>):` reemplaza un documento completo, lo que significa que **el documento original será sustituido por el nuevo**.

, donde:

- **<filtro>:** es un **objeto que especifica el criterio de selección para los documentos que se actualizarán**.

Este parámetro es obligatorio.

Existen diferentes formas de definir un filtro, algunas de las **comparaciones más comunes** son:

- **Igualdad:** para **actualizar un documento específico**, donde se especifica el valor exacto de un campo del documento que se desea actualizar.

Ejemplo: actualizar el primer documento que se encuentre en la colección *personas* donde el campo nombre sea igual a *Juan*, y establecer el campo edad en 30.

```
db.personas.updateOne({nombre: "Juan"}, {$set: {edad: 30}})
```

- **Operadores de comparación:** se pueden utilizar para especificar **filtros más complejos**. Algunos ejemplos de operadores de comparación son:

- **\$eq:** compara si dos valores son **iguales**. Ej.: {nota: {\$eq: 6}}
- **\$ne:** compara si dos valores son **distintos**. Ej.: {nota: {\$ne: 7}}
- **\$gt:** compara si un valor es **mayor** que otro. Ej.: {nota: {\$gt: 6}}
- **\$gte:** compara si un valor es **mayor o igual** que otro.
- **\$lt:** compara si un valor es **menor** que otro.
- **\$lte:** compara si un valor es **menor o igual** que otro. Ej.: {curso: "1D", nota: {\$gte: 7, \$lte: 9}} → Pertenece al curso 1D y la nota está comprendida entre 7 y 9.
- **\$in:** compara si un valor **se encuentra en un conjunto de valores**. Ej.: {nota: {\$in: [5, 7, 8]}}
- **\$nin:** compara si un valor **no se encuentra en un conjunto de valores**.
- **\$exists:** comprobar si el **campo se encuentra en los documentos**. Ej.: {nota: {\$exists: true}}

Ejemplo: actualizar el primer documento que se encuentre en la colección *personas* donde el campo edad es menor que 30, y establecer el campo estado en *"joven"*.

```
db.personas.updateOne({edad: {$lt: 30}}, {$set: {estado: "joven"}})
```

- **Expresiones regulares:** también se pueden utilizar para definir filtros más complejos.

Ejemplo: actualizar el primer documento que se encuentre en la colección *personas* donde el campo nombre comience con la letra *"J"* y establecer el valor del campo estado en *"comienza por J"*.

```
db.personas.updateOne({nombre: /^J/}, {$set: {estado: "comienza por J"}})
```

- **Operadores lógicos:**

- **\$and:** permite **definir un filtro que sea verdadero si se cumplen todas las condiciones**.

Ejemplo: actualizar el primer documento que tengan un campo edad mayor a 18 y un campo sexo igual a *"Femenino"*. Se puede utilizar el filtro:

```
{ $and: [{ edad: { $gt: 18 } }, { sexo: "Femenino" } ] }
```

- **\$or:** permite **definir un filtro que sea verdadero si se cumple al menos una de las condiciones**.

Ejemplo: actualizar el primer documento que tengan un campo edad mayor a 18 o un campo sexo igual a *"Masculino"*. Se puede utilizar el filtro:

```
{ $or: [{ edad: { $gt: 18 } }, { sexo: "Masculino" } ] }
```

- **\$not:** permite **definir un filtro que sea verdadero si no se cumple una condición**.

Ejemplo: actualizar el primer documento que no tengan un campo edad mayor a 18. Se puede utilizar el filtro:

```
{ edad: { $not: { $gt: 18 } } }
```

- **<cambios>:** es un **objeto que especifica los cambios que se deben realizar en el documento seleccionado**.

Este parámetro es obligatorio. Algunos de los **operadores de actualización más comunes** son:

- **\$set:** **establece el valor de un campo en un documento**. Si el campo **no existe en el documento**, **\$set lo agrega**. Ej.: {\$set: {edad: 34}}
- **\$unset:** **elimina un campo** de un documento. Ej.: {\$unset: {edad: 34}}
- **\$inc:** **incrementa el valor** de un campo numérico en una cantidad específica. Ej.: {\$inc: {edad: 1}}
- **\$mul:** **multiplica el valor** de un campo numérico por un número específico.
- **\$min:** **establece el valor de un campo numérico en el menor de dos valores**, el valor actual del campo

- y un valor específico. Ej.: `{ $min: { precio: 25 } }`
 - **\$max**: establece el valor de un campo numérico en el mayor de dos valores, el valor actual del campo y un valor específico. Ej.: `{ $max: { precio: 25 } }`
 - **\$currentDate**: establece el valor de un campo en la fecha y hora actual. Ej.: `{ $currentDate: { ultima_modificacion: true } }`
 - **\$rename**: cambia el nombre de un campo en un documento. Ej.: `{ $rename: { mail: "email" } }`
- **<opciones>**: es un objeto opcional que **especifica las opciones adicionales** para la operación de actualización. Algunas de las opciones más comunes incluyen:
 - **upsert**: si se establece en **true**, se insertará un nuevo documento si no se encuentra ninguno que coincida con el criterio de selección. El valor predeterminado es **false**.
 - **writeConcern**: especifica el **nivel de garantía que se desea para la operación de actualización**. Por ejemplo, si se establece en **majority**, la operación de actualización no se considerará exitosa a menos que se haya confirmado que se ha replicado en la mayoría de los nodos secundarios del clúster.
 - **collation**: especifica las **reglas de comparación para la operación de actualización**. Por ejemplo, se puede especificar un idioma y una regla de comparación sensible a mayúsculas/minúsculas o insensible a las mismas.

En **MongoDB**, las operaciones de actualización **tienen como objetivo una única colección**. Todas las operaciones de escritura en **MongoDB son atómicas al nivel de un solo documento**.

```
db.users.updateMany(
  { age: { $lt: 18 } },
  { $set: { status: "reject" } }
```

← collection
← update filter
← update action

Lo ideal para **modificar un único documento** es **filtrar por el `_id`**. Ej.: `db.personas.updateOne({_id: ObjectId("65906a2e023ae21b9400032f")}, {$set: {nombre: "Jaime"}})`

```
contactos> db.personas.updateOne({'_id': ObjectId("65906a2e023ae21b9400032f")}, {$set: {"nombre": "Jaime"}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Ejemplo: reemplazar un documento completo.

```
db.personas.replaceOne({'_id': ObjectId("65906a2e023ae21b9400032f")}, {nombre: "Jaime", edad: 10})
```

Ejemplo: modificar el apellido1 por **López** del ponente con **idPonente** igual a **ESP004**.

```
db.ponentes.updateOne({idPonente: "ESP004"}, {$set: {apellido1: "López"}})
```

Ejemplo: actualizar un ponente y en caso de que no exista lo inserte.

```
db.ponentes.updateOne({idPonente: "ESP010"}, {$set: {apellido1: "López"}}, {upsert: true})
```

Ejemplo: incrementar el valor del campo **total_gratificaciones** en la colección **ponentes** al primer ponente. En caso de que no exista el campo lo agrega.

```
db.ponentes.updateOne({}, {$inc: {total_gratificaciones: 50.5}})
```

Ejemplo: renombrar el nombre del campo **idPonente** en la colección **ponentes** por **idPon** al primer documento. El campo renombrado aparecerá al final del documento (esto no tiene mayor importancia).

```
db.ponentes.updateOne({}, {$rename: {idPonente: "idPon"}})
```

Ejemplo: cambiar la especialidad del ponente con **idPonente** igual a **ESP004** por **Programación** y agregar un campo **ultima_modificacion** con la fecha y hora actuales.

```
db.ponentes.updateOne(
  {idPonente: "ESP004"},          // Se podrían utilizar operadores lógicos
  {$set: {especialidad: "Programación"},
    $currentDate: {ultima_modificacion: true}
  },
  {upsert: true, // Inserta un nuevo registro si no existe
    // writeConcern: {w: "majority"},
    collation: {locale: "es", strength: 2}          // En español diferenciando acentos,
                                                    // mayúsculas y minúsculas.
  }
)
```

En este ejemplo, se actualizará el primer documento que tenga el campo `idPonente` con el valor `ESP004` y se agregará un nuevo campo que contendrá la fecha y horas actuales en la que se modifica el documento. Las **opciones de strength disponibles** son:

- **1: ignora diacríticos** (acentos, tildes, etc.) **y diferencias entre mayúsculas y minúsculas**. Es la comparación más básica. Por ejemplo, "á", "a", "A" y "Á" (si existiera) se considerarían iguales.
- **2: considera los caracteres base y los diacríticos, pero sigue ignorando las diferencias entre mayúsculas y minúsculas**. "á" y "a" se considerarían diferentes, pero "A" y "a" seguirían siendo iguales.
- **3: considera los caracteres base, los diacríticos y las diferencias entre mayúsculas y minúsculas**. Este es el nivel más preciso. "á", "a", "A" y "Á" (si existiera) serían todos diferentes.

3.11.3 Eliminación.

Las **operaciones de eliminación eliminan documentos de una colección**. *MongoDB* proporciona los siguientes **métodos** para eliminar documentos de una colección:

- `db.<nombre_colección>.deleteOne(<filtro> [, {writeConcern: documento}])`: elimina **un solo documento**, el primer documento que cumple con el criterio de selección especificado en `<filtro>`.
- `db.<nombre_colección>.deleteMany(<filtro> [, {writeConcern: documento}])`: elimina **varios documentos**.

, donde:

- **<nombre_colección>**: es el **nombre de la colección** en la que se desea eliminar el documento.
- **<filtro>**: es un documento que contiene los **criterios de selección para el documento que se desea eliminar**. Este filtro es **similar al que se utiliza en la función `updateOne()`**.
- **writeConcern**: es un **documento opcional que especifica el nivel de confirmación de la escritura**.

En *MongoDB*, las operaciones de eliminación **tienen como objetivo una única colección**. Todas las operaciones de escritura en *MongoDB* son **atómicas al nivel de un solo documento**.

Se pueden especificar criterios o filtros que identifiquen los documentos que se eliminarán. Estos filtros utilizan la misma sintaxis que las operaciones de consulta o actualización.

```
db.users.deleteMany(
  { status: "reject" }
)
```

← collection
← delete filter

Lo ideal para **eliminar un único documento** es **filtrar por el `_id`**. Ej.: `db.personas.deleteOne({"_id": ObjectId("65906a2e023ae21b9400032f")})`

```
contactos> db.personas.deleteOne({"_id": ObjectId("65906a2e023ae21b9400032f")})
{ acknowledged: true, deletedCount: 1 }
```

Ejemplo: eliminar el primer documento de la colección `ponentes` en el que el campo `nombre` es *Silvia*.

```
db.ponentes.deleteOne({nombre: "Silvia"})
```

Se pueden **eliminar varios documentos** de una colección utilizando el comando `db.<nombre_colección>.deleteMany(<filtro>)`. Su funcionamiento es similar a `deleteOne()`.

Ejemplo: eliminar todos los documentos de la colección `ponentes`.

```
db.ponentes.deleteMany({}) // No se especifica ningún filtro {}
```

3.11.4 Consulta.

Las **operaciones de lectura recuperan documentos de una colección**; es decir, consultar una colección de documentos. *MongoDB* proporciona el método `db.<nombre_colección>.find(<filtro> [, <proyección>])` para leer documentos de una colección, donde:

- **<nombre_colección>**: es el nombre de la colección en la que se desea realizar la consulta.
- **<filtro>**: es un **documento que contiene los criterios de selección** de los documentos a obtener. **Si no se indica, devuelve todos los documentos**. Similar a lo visto en el apartado *Actualización*.
- **<proyección>**: es un documento opcional que **especifica los campos que se desean obtener y los campos que se desean excluir**. Si no se especifica una proyección, se devolverán todos los campos del documento.

También se pueden utilizar los comandos `db.<nombre_colección>.findOne(<filtro> [, <proyección>])` o `db.<nombre_colección>.findMany(<filtro> [, <proyección>])` para obtener un solo documento o varios documentos, respectivamente. En cuanto a la sintaxis, **las funciones `find()` y `findOne()` tienen la misma sintaxis**. Por otro lado, **`findMany()` no es una función incorporada en *MongoDB*, por lo que su sintaxis puede variar según la**

implementación específica que se esté utilizando.

Se pueden especificar filtros de consulta o criterios que identifiquen los documentos a devolver.

```
db.users.find(
  { age: { $gt: 18 } },
  { name: 1, address: 1 }
).limit(5)
```

← collection
← query criteria
← projection
← cursor modifier

Además, se pueden establecer **criterios de consulta**. Ej.: `db.personas.find({nombre: "Carlos"})`

Existe la posibilidad de **realizar comparaciones** haciendo uso de `$gt` (>), `$gte` (>=), `$lt` (<), `$lte` (<=), `$eq` (==), `$ne` (<>), `$in` (entre) y `$nin` (no entre). Ejemplo: `db.personas.find({fecha_nacimiento: {$gte: new ISODate("1980-01-01")}}) / db.amigos.find({nota: {$in: [5, 7, 8]}}, {nombre: 1, curso: 1});`

Para **aplicar formato a los resultados de la búsqueda**, se agrega el **método pretty()** al final de la sentencia `db.<nombre_colección>.find(...)`. La sintaxis para **buscar documentos en una colección y aplicar el formato** es la siguiente:

```
db.<nombre_colección>.find(<filtro> [, <proyección>]).pretty()
```

La **sintaxis** de `find()` con el método `sort()` en una colección de *MongoDB* es la siguiente:

```
db.<nombre_colección>.find(<filtro> [, <proyección>]).sort(<criterios_ordenación>)
```

, donde `<criterios_ordenación>` es un **documento que especifica cómo se deben ordenar los documentos encontrados**. Debe incluir un campo y un valor que especifique el orden, por ejemplo, `{<campo>: 1}` ordena los documentos de forma **ascendente** por el campo especificado, mientras que `{<campo>: -1}` los ordena de forma **descendente**. Si no se indica ningún criterio de ordenación los documentos se mostrarán en el orden en el que fueron insertados.

Para **recuperar un número de documentos determinado**, se agrega el **método limit()** al final de la sentencia `db.<nombre_colección>.find(...)`. Por lo tanto, la **sintaxis** es la siguiente:

```
db.<nombre_colección>.find(<filtro> [, <proyección>]).limit(n)
```

, donde `n` es un número entero que **representa el número máximo de documentos que se desean obtener en la consulta**.

Aparte del método `limit()`, hay un método más `skip()` que también acepta argumentos de tipo numérico y se usa para **omitir la cantidad de documentos indicada**. La **sintaxis** es la siguiente:

```
db.<nombre_colección>.find(<filtro> [, <proyección>]).skip(n)
```

3.12 Operaciones de consulta.

Los siguientes **ejemplos utilizan la colección inventario**:

```
db.inventario.insertMany([
  {articulo: "diario", cantidad: 25, tamano: {h: 14, w: 21, um: "cm"}, estado: "A"},
  {articulo: "cuaderno", cantidad: 50, tamano: {h: 8.5, w: 11, um: "in"}, estado: "A"},
  {articulo: "papel", cantidad: 100, tamano: {h: 8.5, w: 11, um: "in"}, estado: "D"},
  {articulo: "planificador", cantidad: 75, tamano: {h: 22.85, w: 30, um: "cm"}, estado: "D"},
  {articulo: "postal", cantidad: 45, tamano: {h: 10, w: 15.25, um: "cm"}, estado: "A"}
])
```

Para **seleccionar todos los documentos de la colección**, pasar un documento vacío como parámetro de filtro de consulta al método de búsqueda (el parámetro de filtro de consulta determina los criterios de selección). Ejemplos:

```
db.inventario.find({})
db.getCollection('inventario').find({})
```

`SELECT * FROM inventory`

Para **especificar condiciones de igualdad**, utilizar expresiones `<campo>:<valor>` en el documento de filtro de consulta (`{<campo1>: <valor1>, ...}`). Ejemplo:

```
db.inventario.find({estado: "D"})
```

`SELECT * FROM inventory WHERE status = "D"`

En *MongoDB*, las consultas a cadenas son sensibles a mayúsculas/minúsculas, por lo que "juan" no coincidirá con "Juan" a menos que se usen expresiones regulares con el modificador `i` para hacerlas insensibles. Por ejemplo:

```
db.inventario.find({estado: {$regex : "d", $options: "i"}})
```

Un documento de filtro de consulta puede utilizar los **operadores de consulta para especificar condiciones** de la siguiente forma: `{<campo1>: {<operador1>: <valor1>}, ...}`. Ejemplo:

```
db.inventario.find({estado: {$in: ["A", "D"]}})
```

`SELECT * FROM inventory WHERE status in ("A", "D")`

Una **consulta compuesta** puede especificar condiciones para varios campos en los documentos de la colección. Implícitamente, las cláusulas de una consulta compuesta están conectadas por una **conjunción lógica AND**, lo que significa que la consulta selecciona los documentos que cumplen con todas las condiciones especificadas.

El siguiente ejemplo recupera todos los documentos de la colección inventario donde estado es igual a "A" y cantidad es menor que 30:

```
db.inventario.find({estado: "A", cantidad: {$lt: 30}})
db.inventario.find({$and: [{estado: "A"}, {cantidad: {$lt: 30}}]})
```

En el siguiente ejemplo se hace uso de **OR** mezclado con AND, el documento de consulta compuesta selecciona todos los documentos de la colección donde estado es igual a "A" y cantidad es menor que 30 o artículo comienza con el carácter 'p':

```
db.inventario.find({
  estado: "A",
  $or: [{cantidad: {$lt: 30}}, {articulo: /^p/}]
})
```

```
SELECT * FROM inventory WHERE status = "A" AND ( qty < 30 OR item LIKE "p%")
```

Para especificar una condición de **consulta en campos de un documento incrustado/anidado**, utilizar la notación de puntos (**campo.campoAnidado**). La siguiente consulta selecciona todos los documentos donde el campo anidado h es menor que 15, el campo anidado um es igual a "in" y el campo estado es igual a "D":

```
db.inventario.find({"tamano.h": {$lt: 15}, "tamano.um": "in", estado: "D"})
```

3.12.1 Consultas y arrays.

Para especificar la **condición de igualdad en un array**, utilizar el **documento de consulta** {<campo>: <valor>} dónde en <valor> está la matriz exacta que debe coincidir, incluido el orden de los elementos. Para los ejemplos de consultas sobre array se utilizará la colección **inventario2**:

```
db.inventario2.insertMany([
  {articulo: "diario", cantidad: 25, tags: ["blank", "red"], dim_cm: [14, 21]},
  {articulo: "cuaderno", cantidad: 50, tags: ["red", "blank"], dim_cm: [14, 21]},
  {articulo: "papel", cantidad: 100, tags: ["red", "blank", "plain"], dim_cm: [14, 21]},
  {articulo: "planificador", cantidad: 75, tags: ["blank", "red"], dim_cm: [22.85, 30]},
  {articulo: "postal", cantidad: 45, tags: ["blue"], dim_cm: [10, 15.25]}
])
```

El siguiente ejemplo consulta todos los documentos donde el valor del campo tags es un array con exactamente dos elementos "red" y "blank" en el mismo orden especificado:

```
db.inventario2.find({tags: ["red", "blank"]})
```

Si, en cambio, se desea encontrar un array que contenga tanto los elementos "red" como "blank", **sin tener en cuenta el orden u otros elementos de la matriz**, utilizar el operador **\$all**:

```
db.inventario2.find({tags: {$all: ["red", "blank"]}})
```

Para **consultar si el campo del array contiene al menos un elemento con el valor especificado** (en cualquier posición), usar el filtro {<campo>: <valor>} donde <valor> está el valor del elemento. El siguiente ejemplo consulta todos los documentos donde tags es un array que contiene la cadena "red" como uno de sus elementos:

```
db.inventario2.find({tags: "red"})
```

Por ejemplo, la siguiente operación consulta todos los documentos donde el array dim_cm contiene al menos un elemento cuyo valor es mayor que 25:

```
db.inventario2.find({dim_cm: {$gt: 25}})
```

Al **especificar condiciones compuestas en elementos de un array**, se puede especificar la consulta de modo que **un solo elemento del array cumpla estas condiciones o cualquier combinación de elementos de la matriz cumpla las condiciones**.

El siguiente ejemplo consulta documentos donde el array dim_cm contiene **elementos que en alguna combinación satisfacen las condiciones de la consulta**; por ejemplo, un elemento puede satisfacer la condición mayor que 15 y otro elemento puede satisfacer la condición menor que 20, o un solo elemento puede satisfacer ambas:

```
db.inventario2.find({dim_cm: {$gt: 15, $lt: 20}})
```

Utilizar el operador **\$elemMatch** para **especificar múltiples criterios en los elementos de una matriz de modo que**

al menos un elemento de la matriz satisfaga todos los criterios especificados.

El siguiente ejemplo consulta documentos donde el array `dim_cm` contiene al menos un elemento que es mayor que 22 y menor que 30:

```
db.inventario2.find({dim_cm: {$elemMatch: {$gt: 22, $lt: 30}}})
```

Usando la notación de puntos, se puede **especificar condiciones de consulta para un elemento en un índice o posición particular de un array**. El array utiliza indexación de base cero.

El siguiente ejemplo consulta todos los documentos donde el segundo elemento de `dim_cm` es mayor que 25:

```
db.inventario2.find({"dim_cm.1": {$gt: 25}})
```

Utilizar el operador `$size` para **consultar arrays por número de elementos**. Por ejemplo, lo siguiente selecciona documentos donde el array `tags` tiene 3 elementos:

```
db.inventario2.find({"tags": {$size: 3}})
```

3.12.2 Selección de campos a devolver en las consultas.

Si no se especifica un documento de proyección, el método `db.<nombre_colección>.find()` devuelve todos los campos de los documentos coincidentes.

Para limitar la cantidad de datos que *MongoDB* envía a las aplicaciones, **puede incluirse un documento de proyección para especificar o restringir los campos a devolver**.

Una **proyección puede incluir explícitamente varios campos estableciendo el campo a valor 1 en el documento de proyección**. El siguiente ejemplo devuelve todos los documentos que coinciden con la consulta y en el conjunto de resultados, solo los campos `articulo`, `estado` y por defecto `_id` se devuelven en los documentos coincidentes:

```
db.inventario.find({estado: "A"}, {articulo: 1, estado: 1})
```

Se puede **eliminar el campo `_id` de los resultados poniéndolo a 0 en la proyección**, como en el siguiente ejemplo:

```
db.inventario.find({estado: "A"}, {articulo: 1, estado: 1, _id: 0})
```

En lugar de enumerar los campos que se devolverán en el documento coincidente, **se puede utilizar una proyección para excluir campos específicos**. El siguiente ejemplo devuelve todos los campos excepto los campos `estado` e `instock` en los documentos coincidentes:

```
db.inventario.find({estado: "A"}, {estado: 0, instock: 0})
```

Se pueden **devolver campos específicos en un documento incrustado. Utilizar la notación de puntos para hacer referencia al campo incrustado y configurarlo en 1 en el documento de proyección**. El siguiente ejemplo devuelve `_id`, `articulo`, `estado` y `um` del documento tamaño:

```
db.inventario.find(
  {estado: "A"},
  {articulo: 1, estado: 1, "cantidad.um": 1}
)
```

3.12.3 Consulta de campos nulos o faltantes.

Los distintos operadores de consulta en *MongoDB* tratan los valores `null` de forma diferente.

Para los ejemplos de este punto se utilizará la colección `inventario3`:

```
db.inventario3.insertMany([
  { _id: 1, articulo:
    null },
  { _id: 2 }
```

El filtro `{<campo>: null}` coincide con los documentos que contienen el campo `<campo>` con valor `null` o no contienen el campo `<campo>`. Ejemplo:

```
db.inventario3.find({articulo: null})
```

El filtro `{<campo>: {$type: 10}}` coincide sólo con documentos que contienen el campo `<campo>` cuyo valor es `null`; es decir, el valor de `<campo>` es de tipo *BSON Null* (tipo *BSON* 10):

```
db.inventario3.find({articulo: {$type: 10}})
```

El filtro `{<campo>: {$exists: false}}` coincide con los documentos que no contienen el campo `<campo>`. El siguiente ejemplo consulta documentos que no contienen un campo:

```
db.inventario3.find({articulo: {$exists: false}})
```

3.13 Agregaciones y tuberías.

Las operaciones de agregación **ayudan a procesar documentos y retornar resultados calculados**. Se utilizan mayoritariamente para: agrupar valores de varios documentos, procesamiento y operaciones matemáticas, analizar cambios de datos a lo largo del tiempo.

A su vez, las operaciones de agregaciones **se pueden realizar de dos formas principales**:

3.13.1 Métodos de agregación de propósito único.

Los métodos de agregación de propósito único son métodos que se aplican en una sola colección. Los métodos son simples, pero carecen de las capacidades de tuberías.

- **Contar documentos: `countDocuments`**. Para contar documentos se utiliza el método `countDocuments`, este simplemente retorna un recuento del número de documentos existentes.

```
db.<nombre_colección>.countDocuments([<filtro> [, <opciones>]])
```

- **Obtención de documentos distintos**. Para obtener una matriz de documentos que tienen valores distintos, se utiliza el método `distinct`, este método debe recibir el campo especificado.

```
db.<nombre_colección>.distinct(<campo> [, <filtro>] [, <opciones>]])
```

Ejemplos:

```
db.inventario.countDocuments()
db.ponentes.countDocuments({edad: {$gt: 30}})
db.inventario.distinct("estado")
db.ponentes.distinct("nombre", {especialidad: "Bases de Datos"})
```

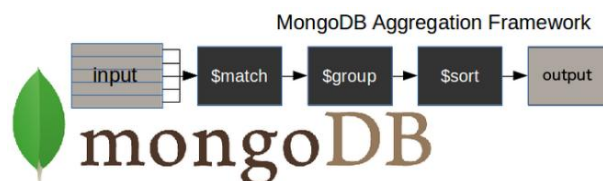
3.13.2 Tuberías y transformaciones.

Para realizar el procesamiento de documentos, *MongoDB* se basa en el “Patrón de filtro de tubería”, utilizado comúnmente en arquitecturas de software y sistemas operativos *Unix*.

Este patrón consta de una o más etapas, en donde cada etapa realiza una operación con los datos de entrada. A su vez, la salida o resultado se la entrega a la siguiente etapa para su procesamiento.

Para aplicar este patrón, *MongoDB* dispone del **método `aggregate()`**. Este método **recibe un array de operaciones** (más conocidas como tuberías) **las cuales se ejecutarán en secuencia una por una hasta llegar al resultado esperado**.

Ejemplo de funcionamiento:



Su **sintaxis** es:

```
db.nombreColección.aggregate(<pipeline> [, <opciones>])
```

, donde:

- **<pipeline>**: lista de **etapas** (u operaciones) **de agregación que se ejecutan en orden**. Cada etapa es un objeto que describe una operación de agregación específica, como filtrar documentos, agrupar documentos, proyectar campos u ordenar resultados entre otras. Cada etapa recibe los documentos de la etapa anterior como entrada y produce una salida que se convierte en entrada para la siguiente etapa en la secuencia. Las etapas de agregación más comunes son:
 - **\$match: filtrar documentos**. Filtra los resultados para que solo estén los que cumplen ciertos criterios. Se puede filtrar antes o después de agregar los resultados según se defina.
 - **\$group: agrupar documentos**. Agrupa documentos según compartan el valor de uno o varios atributos. Realiza operaciones de agregación de cada grupo. Algunas de las funciones que usa son (ya vistas):
 - **\$sum**: calcula la **suma de los valores** de un campo específico en todos los documentos que cumplen con los criterios de la etapa anterior.

- **\$avg**: calcula el **promedio de los valores** de un campo específico en todos los documentos que cumplen con los criterios de la etapa anterior.
- **\$max**: devuelve el **valor máximo** de un campo específico en todos los documentos que cumplen con los criterios de la etapa anterior.
- **\$min**: devuelve el **valor mínimo** de un campo específico en todos los documentos que cumplen con los criterios de la etapa anterior.
- **\$count**: devuelve el **número de documentos** que cumplen con los criterios de la etapa anterior.
- **\$first**: devuelve el **primer valor del grupo** (según el orden de entrada).
- **\$last**: devuelve el **último valor del grupo** (según el orden de entrada).
- **\$project**: **seleccionar campos** específicos del documento.
- **\$sort**: **ordenar** documentos.
- **\$limit**: **limitar la cantidad de documentos** de salida.
- **\$skip**: **saltar una cierta cantidad de documentos** en la salida.
- **\$unwind**: **descompone un campo array** en múltiples documentos, uno por cada elemento.
- **\$lookup**: realiza una **operación de JOIN entre dos colecciones** (se verá más adelante en la unidad).
- **<opciones>**: **objeto que puede contener opciones adicionales para la operación de agregación**, como especificar un índice para la operación o controlar el tamaño del búfer de resultados. Algunas de las opciones disponibles son:
 - **allowDiskUse**: si se establece en true, permite que los datos intermedios se escriban en el disco en lugar de la memoria, lo que puede ayudar a evitar que se agote la memoria en el servidor.
 - **collation**: especifica reglas de comparación de caracteres para ordenar y comparar cadenas en la operación de agregación. Ej.: {collation: {locale: "es"}}
 - **explain**: si se establece en true, devuelve información detallada sobre cómo se ha realizado la operación de agregación. Ej.: {explain: true}
 - **maxTimeMS**: establece un límite de tiempo máximo en milisegundos para que la operación de agregación se ejecute antes de que se agote el tiempo y se interrumpa. Ej.: {maxTimeMS: 5000}
 - **readConcern**: especifica el nivel de consistencia que se requiere para los datos de entrada durante la operación de agregación.
 - **readPreference**: especifica la preferencia de lectura que se utilizará para leer los datos de entrada durante la operación de agregación.
 - **writeConcern**: especifica el nivel de garantía que se requiere para la escritura de los datos resultantes de la operación de agregación.

Los operadores más utilizados en tuberías son:

- **Filtrado de documentos con criterios**. Para filtrar los documentos que coinciden con las condiciones especificadas se utiliza el operador **\$match**.

```
db.<nombre_colección>.aggregate([{$match: {<campo>: <valor>}}])
```

Ejemplo: db.inventario.aggregate([{\$match: {estado: "D"}}])

- **Orden de documentos**. Para ordenar todos los documentos de entrada y retornar la canalización, se utiliza el operador **\$sort**. Este puede ordenar de forma **ascendente (1)** o **descendente (-1)** según el criterio.

```
db.<nombre_colección>.aggregate([{$sort: {<campo>: <orden>}}])
```

Ejemplo: db.inventario.aggregate([{\$sort: {artículo: 1}}])

- **Selección de campos en específico**. Para seleccionar campos específicos y descartar otros, se utiliza el operador **\$project**. Los valores que puede tomar <mostrar> es 0 (no mostrar) o 1 (mostrar).

```
db.<nombre_colección>.aggregate([{$project: {<campo>: <mostrar>, ... }}])
```

Ejemplo: db.inventario.aggregate([{\$project: {artículo: 1, estado: 1, _id: 0}}])

- **Agrupación de documentos**. Para **separar los documentos en grupos según una "clave de grupo"**. Se utiliza el operador **\$group** y el resultado es un documento para cada clave de grupo única, similar a **GROUP BY** en SQL.

```
db.<colección>.aggregate([{$group: {_id: "$<campo_grupo>", <campo_result>: {$func: "$<campo>"}}]])
```

Si se desea hacer **sobre toda la colección sin agrupar por ningún campo específico**:

```
db.<colección>.aggregate([{$group: {_id: null, <campo_result>: {$func: "$<campo>"}}]])
```

Las **funciones de agregado** que se pueden utilizar son: avg, sum, max, min, first, last y count.

Ejemplos: db.inventario.aggregate([{\$group: {_id: null, num_documentos: {\$count: {}}}}])

```

db.inventario.aggregate([{$group: {_id: "$estado", num_documentos: {$count: {}}}}])
db.inventario.aggregate([{$group: {_id: "$estado", cant_media_por_espado: {$avg:
                                                                    "$cantidad"}}}}])
db.inventario.aggregate([{$group: {_id: "$estado", cant_total_por_espado: {$sum:
                                                                    "$cantidad"}}}}])

```

Ejemplo: con varias etapas.

```

db.ventas.aggregate([
  {$match: {fecha: {$gte: ISODate("2024-01-01"), $lt: ISODate("2025-01-01")}}},
  {$group: {_id: "$producto", totalVentas: {$sum: "$cantidad"}, totalIngresos: {$sum: {$multiply: ["$precio",
                                                                    "$cantidad"]}}}},
  {$sort: {totalVentas: -1}}
])

```

3.14 Funciones.

Función Aritmética	Descripción
\$abs	Valor absoluto de un número
\$add	Añade números a una cantidad o a una fecha (mseg)
\$ceil	Entero menor, mayor o igual que el número especificado
\$divide	Resultado de dividir el primer número por el segundo (2 argumentos)
\$floor	Entero mayor, menor o igual que el número especificado
\$mod	Resto de dividir el primer número por el segundo (2 argumentos)
\$multiply	Multiplica varios números (acepta varios argumentos)
\$pow	Eleva un número a la potencia especificada
\$sqrt	Raíz cuadrada
\$subtract	Resultado de restar dos números (si son fechas, devuelve mseg)
\$trunc	Trunca un número

Función de Cadenas	Descripción
\$concat	Concatena varias cadenas
\$substr	Subcadena de una cadena, desde una posición hasta otra
\$toLowerCase	Convierte una cadena a minúsculas
\$toUpperCase	Convierte una cadena a mayúsculas
\$strcasecmp	Compara dos cadenas. Devuelve 0 si son iguales, 1 si la primera cadena es mayor y -1 si la primera cadena es menor

Función de Grupo	Descripción
\$sum	Suma de valores numéricos, el resto los ignora
\$avg	Media de valores numéricos, el resto los ignora
\$first	Primer valor del grupo
\$last	Último valor del grupo
\$max	Valor máximo de un grupo o array
\$min	Valor mínimo de un grupo o array

Función de Fecha	Descripción
\$dayOfYear	Día del año (número entre 1 y 366)
\$dayOfMonth	Día del mes (número entre 1 y 31)
\$dayOfWeek	Día de la semana (número entre 1 - domingo y 7 - sábado)
\$year	Año, con formato yyyy
\$month	Número del mes (entre 1 - enero y 12 - diciembre)

\$hour	Hora, entre 0 y 23
\$minute	Minutos entre 0 y 59
\$second	Segundos entre 0 y 60
\$dateToString	Fecha en formato String

3.15 Índices.

De forma genérica, **un índice es una estructura de datos utilizada en BD para mejorar la eficiencia de las consultas**. Su propósito principal es permitir un acceso rápido a la información almacenada. Los índices ayudan a acelerar la recuperación de datos relevantes al evitar la necesidad de recorrer todos los registros o elementos dentro de una BD.

Cuando se crea un índice en *MongoDB*, **se genera una estructura de datos que almacena los valores de los campos indexados junto con referencias a los documentos que los contienen**. Esto permite que *MongoDB* encuentre rápidamente los documentos relevantes cuando se realizan consultas basadas en los campos indexados.

Los índices en *MongoDB* **se pueden crear de forma explícita** mediante comandos específicos, **o se pueden generar automáticamente** utilizando índices “de fondo” para los campos definidos como únicos o claves primarias.

A su vez, los índices en *MongoDB* **pueden ser simples**, cuando se crea un índice en un solo campo, **o compuestos**, cuando se crea un índice en múltiples campos combinados. Los índices compuestos permiten optimizar consultas que involucren múltiples condiciones en diferentes campos.

Consideraciones sobre los índices: es importante tener en cuenta que los índices en *MongoDB* ocupan espacio en disco y pueden afectar el rendimiento de las operaciones de escritura, ya que los índices también deben actualizarse cada vez que se modifiquen los datos indexados. Por lo tanto, se debe tener en cuenta el equilibrio entre el rendimiento de las consultas y el impacto en las operaciones de escritura al diseñar y utilizar índices en *MongoDB*.

3.15.1 Método explain.

En *MongoDB*, **explain** es un **método que se utiliza para obtener información detallada sobre cómo se ejecuta una consulta en la BD**. Proporciona una descripción completa del plan de ejecución de la consulta, incluyendo el índice utilizado, el número de documentos escaneados, el tiempo de ejecución y otras estadísticas relevantes.

Al utilizar el método **explain**, se puede obtener información valiosa sobre el rendimiento de una consulta y cómo se está utilizando el índice. Esto permite optimizar las consultas y los índices para mejorar la eficiencia de la BD.

Ejemplo:

```
db.inventario.find({estado: "D"}).explain()
db.inventario.find({estado: "D"}).explain("executionStats").executionStats
```

3.15.2 Creación de un índice.

En *MongoDB*, se utiliza la **función createIndex()** para crear índices en una colección. Su **sintaxis** básica es:

```
db.<nombre_colección>.createIndex(<campo_o_campos> [, <opciones>])
```

, donde:

- **<nombre_colección>**: es el **nombre de la colección** en la que se desea crear el índice.
- **<campo_o_campos>**: es un objeto que especifica el **nombre del campo o los campos** (en este caso, separados por comas) **en los que se desea crear el índice**, cada uno de ellos junto al **valor 1 (ascendente) o -1 (descendente)**. Ej.: {nombre: 1, apellido1: -1} → índice ascendente por el campo *nombre* y descendente por el campo *apellido1*.
- **<opciones>**: es un objeto que especifica **opciones adicionales para el índice**, como la unicidad o el nombre. Ej.: {unique: true, name: "idx_articulo_asc"}

La ventaja de utilizar un índice ascendente es que es eficiente para buscar valores mínimos o encontrar los valores más bajos en una consulta. Por ejemplo, si se realiza una consulta que busca los empleados de menor edad, el índice ascendente puede ser utilizado eficientemente para encontrar rápidamente los valores más bajos en el índice.

En contraste, un índice descendente ordena los valores en orden decreciente, desde el valor más alto hasta el valor más bajo. Este tipo de índice es eficiente para buscar valores máximos o encontrar los valores más altos en una consulta.

Los **índices compuestos** no pueden estar formados por más de 32 campos.

Ejemplo: crear un índice sobre la propiedad estado de la colección inventario.

```
db.inventario.createIndex({estado: 1})
```

En este caso se especifica el campo en el que se creará el índice, y 1 indica que se creará un índice ascendente en ese campo. Se puede cambiar 1 por -1 si se desea un índice descendente.

Ejemplo: `db.inventario.createIndex({articulo: 1, estado: 1})`

Ejemplo: crear un índice único por el campo articulo.

```
db.inventario.createIndex({articulo: 1 }, {unique: true})
```

Ejemplo: crear un índice asignándole un nombre en concreto.

```
db.inventario.createIndex({articulo: 1 }, {name: "idx_articulo_asc"})
```

3.15.3 Eliminación de un índice.

En *MongoDB*, se utiliza la **función dropIndex()** para eliminar un índice de una colección. La **sintaxis** básica de esta función es la siguiente:

```
db.<nombre_colección>.dropIndex(<nombre_del_índice>)
```

, donde:

- **<nombre_colección>**: es el **nombre de la colección** en la que se encuentra el índice.
- **<nombre_del_índice>**: es el **nombre del índice** que se desea eliminar.

El **nombre del índice en MongoDB** generalmente se compone del nombre del campo seguido de un número que indica el tipo de índice (en este caso, "1" para ascendente). Se puede **obtener un listado con los nombres de los índices existentes en una colección** utilizando el método `getIndexes()`:

```
db.<nombre_colección>.getIndexes()
```

Es importante destacar que **algunos índices, como los índices creados automáticamente para el campo `_id`, no pueden ser eliminados**. En ese caso, se generará un error si se intenta utilizar `dropIndex()` para eliminar el índice. También es posible **eliminar varios índices a la vez utilizando la función `dropIndexes()`**, que acepta como parámetro un objeto con opciones adicionales para especificar qué índices se deben eliminar.

Ejemplo: eliminar el índice creado sobre el campo estado ascendente.

```
db.inventario.dropIndex("estado_1")
```

Ejemplo: eliminar todos los índices asociados a la colección inventario.

```
db.inventario.dropIndexes()
```

Ejemplo: eliminar todos los índices asociados a la colección inventario que no sean el índice predeterminado `_id`.

```
db.inventario.dropIndexes({index: {_id: 0}})
```

3.16 Modelado de datos.

El modelado de *BD* en *MongoDB* es **diferente del modelo relacional tradicional**. En *MongoDB*, se utiliza un **modelo de datos no relacional basado en documentos**, lo que significa que **los datos se almacenan en documentos JSON en lugar de en tablas con filas y columnas**. Esto permite una **mayor flexibilidad en la estructura de los datos** y permite la capacidad de almacenar datos anidados y *arrays*.

El modelado de *BD MongoDB* implica **diseñar la estructura de los documentos y colecciones de la BD**, teniendo en cuenta las necesidades de la aplicación y los requisitos de rendimiento.

Algunos **consejos para el modelado** de *BD* en *MongoDB* son:

- ✓ **Desnormalizar los datos**: a diferencia del modelo relacional, donde la normalización es una buena práctica, en *MongoDB* se recomienda desnormalizar los datos **para reducir el número de consultas y mejorar el rendimiento**.
- ✓ **Diseñar el modelo en función de las consultas**: en lugar de diseñar el modelo en función de las tablas y las relaciones como en el modelo relacional, se recomienda diseñar el modelo en función de las consultas **que se realizarán con mayor frecuencia**.

- ✓ **Utilizar referencias y embebidos de manera estratégica:** en *MongoDB*, se pueden utilizar referencias y documentos embebidos para modelar **relaciones entre documentos**. Es importante utilizarlos de manera estratégica para optimizar el rendimiento.
- ✓ **Evitar colecciones con demasiados documentos:** en *MongoDB*, las colecciones pueden contener muchos documentos, pero es recomendable evitar colecciones con demasiados documentos, ya que esto puede afectar el rendimiento.
- ✓ **Utilizar índices de manera adecuada:** los índices pueden mejorar el rendimiento de las consultas en *MongoDB*, por lo que es importante utilizarlos de manera adecuada y estratégica.

Para **modelar una BD MongoDB**, se deben seguir algunos **pasos generales**:

- ✓ **Identificar las entidades clave de la aplicación y sus relaciones:** se debe comprender la estructura de los datos de la aplicación y cómo se relacionan entre sí.
- ✓ **Diseñar la estructura de la colección:** cada colección en *MongoDB* representa un conjunto de documentos relacionados. Se debe **decidir qué campos incluir en cada documento y cómo se deben anidar**.
- ✓ **Optimizar el rendimiento:** se deben considerar las consultas que se ejecutarán con mayor frecuencia y diseñar la estructura de la colección para que esas consultas sean rápidas y eficientes.
- ✓ **Implementar la validación de datos:** se pueden utilizar las reglas de validación de *MongoDB* para garantizar que los datos almacenados en la *BD* cumplan con ciertas condiciones.

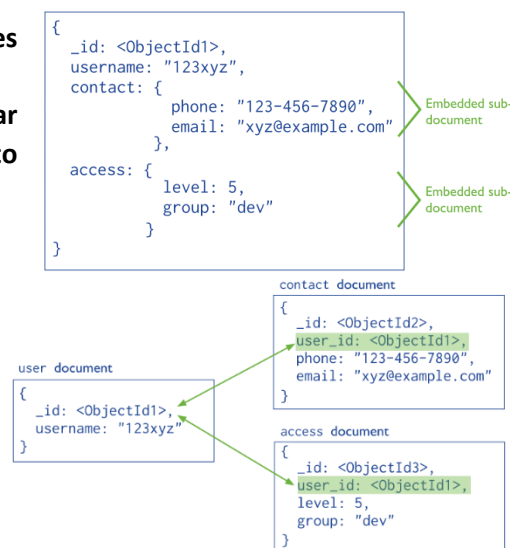
En general, el modelado de *BD MongoDB* es un proceso más flexible y menos estructurado que el modelado de *BD relacionales*. Sin embargo, es importante comprender las necesidades de la aplicación y diseñar la estructura de la *BD* de manera adecuada para garantizar el rendimiento y la integridad de los datos.

Los **pasos** para convertir un diagrama entidad-relación (*DER*) en un modelo orientado a documentos son:

- ✓ **Crear colecciones para cada entidad:** las **entidades del diagrama E/R** se convierten en **colecciones de documentos**. Cada documento representa una instancia de la entidad y contiene todos sus campos y valores.
- ✓ **Ajustar las relaciones entre entidades:** se pueden utilizar **diferentes enfoques para modelar las relaciones**, como **embebidos, referencias o una combinación de ambos**. Por ejemplo, si una entidad tiene una relación uno-a-muchos con otra entidad, se puede embeber documentos de la entidad secundaria dentro de la entidad principal o utilizar referencias para conectar los documentos de ambas colecciones.

Las **relaciones en MongoDB** se pueden modelar con dos enfoques diferentes:

- ✓ **Relación incrustada/embebida:** relación que implica **almacenar documentos secundarios incrustados dentro de un documento principal**.
- ✓ **Relación referenciada:** relación que implica **conectar colecciones diferentes por medio de referencias**. La referencia se realiza a través de una propiedad o clave en común.



Cabe destacar que, en *MongoDB*, **no existe una restricción de integridad referencial** como la que se encuentra en una *BD relacional*. Por lo tanto, es importante asegurarse de que las referencias a otras entidades sean coherentes y estén actualizadas adecuadamente.

Ejemplo: modelado de una *BD* para una aplicación que almacena información sobre productos, categorías y órdenes de compra.

- ❖ **Productos:** la colección de productos contiene información sobre los productos que se venden. Cada documento en la colección representa un producto y puede tener los siguientes campos:

```

{
  _id: ObjectId,
  nombre: String,
  descripcion: String,
  precio: Number,

```

```
categoria: ObjectId }
```

, donde:

- `_id`: identificador único del producto.
- `nombre`: nombre del producto.
- `descripcion`: descripción del producto.
- `precio`: precio del producto.
- `categoria`: identificador de la categoría a la que pertenece el producto.

- ❖ **Categorías**: la colección de categorías contiene información sobre las categorías de productos. Cada documento en la colección representa una categoría y puede tener los siguientes campos:

```
{ _id: ObjectId,
  nombre: String,
  descripcion: String }
```

, donde:

- `_id`: identificador único de la categoría.
- `nombre`: nombre de la categoría.
- `descripcion`: descripción de la categoría.

- ❖ **Órdenes de compra**: la colección de órdenes de compra contiene información sobre las órdenes de compra realizadas por los clientes. Cada documento en la colección representa una orden y puede tener los siguientes campos:

```
{ _id: ObjectId,
  productos: [{
    producto: ObjectId,
    cantidad: Number,
    precio: Number
  }],
  total: Number,
  cliente: String,
  fecha: Date }
```

, donde:

- `_id`: identificador único de la orden de compra.
- `productos`: un array que contiene información sobre los productos comprados en la orden, incluyendo el identificador del producto, la cantidad comprada y el precio unitario del producto en el momento de la compra.
- `total`: el costo total de la orden.
- `cliente`: nombre del cliente que realizó la compra.
- `fecha`: fecha en que se realizó la compra.

Ejemplo: modelado de una *BD* para una aplicación de redes sociales en la que se comparten fotos y se hace el seguimiento de amigos.

- ❖ **Colección usuarios**:

```
{ _id: ObjectId("614f5c5fa33e481d5c7e04ae"),
  nombre: "Juan Pérez",
  correo: "juan.perez@gmail.com",
  contraseña: "hash_de_la_contraseña",
  amigos: [ObjectId("614f5c5fa33e481d5c7e04af"), ObjectId("614f5c5fa33e481d5c7e04b0")],
  seguidores: [ObjectId("614f5c5fa33e481d5c7e04b1"), ObjectId("614f5c5fa33e481d5c7e04b2")],
  publicaciones: [ObjectId("614f5c5fa33e481d5c7e04b3"), ObjectId("614f5c5fa33e481d5c7e04b4")] }
```

En esta colección, se almacenan los datos de los usuarios. Cada documento representa un usuario y contiene campos para su nombre, correo electrónico, contraseña y listas de amigos, seguidores y publicaciones.

- ❖ **Colección publicaciones**:

```
{ _id: ObjectId("614f5c5fa33e481d5c7e04b3"),
  autor: ObjectId("614f5c5fa33e481d5c7e04ae"),
  texto: "¡Acabo de subir una nueva foto a mi perfil!",
  imagen: "ruta/de/la/imagen.jpg",
  fecha_creacion: ISODate("2021-09-26T12:34:56Z"),
  me_gusta: [ObjectId("614f5c5fa33e481d5c7e04b1"), ObjectId("614f5c5fa33e481d5c7e04b2")],
  comentarios: [
    { autor: ObjectId("614f5c5fa33e481d5c7e04b0"),
      texto: "¡Qué buena foto!",
      fecha_creacion: ISODate("2021-09-27T10:11:12Z") },
  ] }
```

```

    { autor: ObjectId("614f5c5fa33e481d5c7e04af"),
      texto: "Me encanta esa vista",
      fecha_creacion: ISODate("2021-09-27T14:15:16Z") }
  ]
}

```

En la colección publicaciones, se almacenan las publicaciones que los usuarios hacen en la aplicación. Cada documento representa una publicación y contiene campos para el autor, el texto, la imagen (opcional), la fecha de creación, la lista de usuarios a quienes les gusta la publicación y la lista de comentarios en la publicación. Con este ejemplo simple, se puede ver cómo modelar una *BD* en *MongoDB* utilizando documentos *JSON* que contienen información relacionada.

Ejemplo: considerar el siguiente modelo *E/R* simplificado para una aplicación de seguimiento de tareas.

Entidades y atributos:

- ✓ *Usuario*: *id_usuario*, *nombre_usuario*, *email*.
- ✓ *Proyecto*: *id_proyecto*, *nombre_proyecto*, *descripcion_proyecto*.
- ✓ *Tarea*: *id_tarea*, *descripcion_tarea*, *fecha_inicio*, *fecha_fin*.

Relaciones:

- ✓ Tarea pertenece a un proyecto.
- ✓ Tarea es asignada a un usuario.

Para convertir el modelo *E/R* en un modelo orientado a documentos, se podrían seguir los siguientes pasos:

- ✓ Cada entidad se convierte en una colección de documentos.
- ✓ Cada atributo se convierte en un campo dentro de los documentos de la colección correspondiente.
- ✓ Las relaciones se manejan utilizando referencias.

El modelo orientado a documentos podría quedar de la siguiente forma:

▪ Colección usuarios:

```

{ _id: ObjectId("615474a6d407c6f98a6f1a6d"),
  nombre_usuario: "Juan",
  email: "juan@ejemplo.com" }

```

▪ Colección proyectos:

```

{ _id: ObjectId("615474a6d407c6f98a6f1a6e"),
  nombre_proyecto: "Proyecto 1",
  descripcion_proyecto: "Descripción del proyecto 1" }

```

▪ Colección tareas:

```

{ _id: ObjectId("615474a6d407c6f98a6f1a6f"),
  descripcion_tarea: "Tarea 1",
  fecha_inicio: ISODate("2024-01-01T00:00:00.000Z"),
  fecha_fin: ISODate("2024-01-31T23:59:59.000Z"),
  id_proyecto: ObjectId("615474a6d407c6f98a6f1a6e"),
  id_usuario: ObjectId("615474a6d407c6f98a6f1a6d") }

```

En el modelo orientado a documentos, cada documento representa una instancia de una entidad y contiene todos sus campos y valores. Las relaciones se manejan utilizando referencias a los identificadores únicos de los documentos relacionados en otras colecciones.

En este ejemplo, la colección de tareas contiene referencias a los identificadores de los documentos de las colecciones de proyectos y usuarios. De esta forma, es posible realizar consultas y búsquedas en todas las colecciones relacionadas utilizando referencias y la sintaxis adecuada en las consultas *MongoDB*.

Ejemplo: suponer que se tiene un modelo *E/R* para una tienda que incluye las entidades *Cliente*, *Pedido*, *Producto* y *Categoría*. La relación entre ellas es que un cliente realiza uno o más pedidos, cada pedido puede contener uno o más productos, y cada producto puede pertenecer a una o más categorías.

Para convertir este modelo *E/R* a un modelo orientado a documentos, se podría diseñar una estructura de documentos que represente cada entidad como una colección, y que incluya referencias a otras colecciones donde sea necesario. Por ejemplo:

▪ Colección clientes:

```

{ _id: ObjectId("123456789012345678901234"),
  nombre: "Juan",
  apellidos: "Pérez",

```

```
email: "juan.perez@ies.com",
pedidos: [ ObjectId("123456789012345678901231"),
           ObjectId("123456789012345678901232") ] }
```

- Colección pedidos:

```
{ _id: ObjectId("123456789012345678901231"),
  fecha: ISODate("2023-03-31T13:30:00Z"),
  productos: [ { producto_id: ObjectId("123456789012345678901221"),
                cantidad: 2 },
               { producto_id: ObjectId("123456789012345678901222"),
                cantidad: 1 } ] }
```

- Colección productos:

```
{ _id: ObjectId("123456789012345678901221"),
  nombre: "Camiseta",
  precio: 19.99,
  categorias: [ ObjectId("123456789012345678901211"),
                ObjectId("123456789012345678901212") ] }
```

- Colección categorías:

```
{ _id: ObjectId("123456789012345678901211"),
  nombre: "Ropa",
  descripcion: "Prendas de vestir" }
```

En estos documentos, se utilizan los ObjectId's generados por *MongoDB* para establecer las referencias entre colecciones. En el documento de cliente se hace referencia a los pedidos que ha realizado utilizando los ObjectId.

Ejemplo:

```
//Colección emple, con 4 empleados.
db.emple.drop();
db.emple.insert({_id:'empl',nombre:"Juan", salario:1000, fechaalta:"10/10/1999"})
db.emple.insert({_id:'emp2',nombre:"Alicia", salario:1400, fechaalta:"07/08/2000", oficio: "Profesora"})
db.emple.insert({_id:'emp3',nombre:"María Jesús", salario:1500, fechaalta: "05/01/2005", oficio: "Analista", comision:100})
db.emple.insert({_id:'emp4',nombre:"Alberto", salario:1100, fechaalta:"15/11/2001"})
db.emple.find();

//Colección depart con dos departamentos, asignamos los dos primeros
//empleados al primer departamento, y al segundo el resto de emple
db.depart.drop();
db.depart.insert({_id:'depl',nombre:"Informática",loc:'Madrid', emple:['empl', 'emp2']})
db.depart.insert({_id:'dep2',nombre:"Gestión",loc:'Talavera', emple:['emp3', 'emp4' ]})
db.depart.find()
```

3.17 Operador *Lookup*.

Para **consultar documentos enlazados en varias colecciones**, es posible utilizar la funcionalidad de agregación en *MongoDB* y el **operador \$lookup**.

Es importante tener en cuenta que **estas consultas pueden ser bastante complejas** dependiendo de la cantidad de datos y la estructura de las colecciones, por lo que es recomendable hacer uso de índices y optimizaciones para mejorar el rendimiento.

Sintaxis: {

```
  $lookup: {
    from: <colección_destino>, // Colección desde la cual se obtendrán los documentos para unir
    localField: <campo_local>, // Campo colección actual a utilizar para realizar la unión
    foreignField: <campo_destino>, // Campo colección destino a utilizar para realizar unión
    as: <nombre_del_campo_salida> // Nombre del nuevo campo que contendrá los documentos unidos
  }
}
```

Ejemplo: suponer que se tienen dos colecciones usuarios y pedidos, y que la colección pedidos tiene un campo `id_usuario` que hace referencia al campo `_id` de la colección usuarios. Se quiere obtener una lista de todos los usuarios con sus respectivos pedidos.

```
db.usuarios.aggregate([
  { $lookup: {
    from: "pedidos",
    localField: "_id",
    foreignField: "id_usuario",
    as: "pedidos" } } ] )
```

```
db.ventas.aggregate([
  {
    $lookup: {
      from: "productos",
      localField: "producto_id",
      foreignField: "_id",
      as: "producto"
    }
  }
])
```


3.18 Aplicar reglas de validación de esquema en una colección.

Como se ha visto anteriormente, un documento de *MongoDB* puede almacenar documentos de cualquier tamaño de pares clave-valor, así como documentos anidados.

Sin embargo, en este punto, **se quiere imponer que una colección contenga documentos de estrictamente las mismas propiedades** (claves). Así, por ejemplo, se quiere que cada documento tenga exactamente las mismas propiedades/campos.

En *MongoDB*, la validación de esquemas **se realiza utilizando el Validador de Esquemas**. Este validador **permite definir y aplicar reglas de validación** para los documentos que se almacenan en una colección.

Para habilitar la validación de esquemas en *MongoDB*, primero **es necesario crear un esquema utilizando el formato JSON Schema**. Este esquema **puede especificar restricciones para los campos**, como el tipo de datos permitido, la longitud máxima o mínima del valor y si el campo es requerido o no.

Una vez creado el esquema, **se puede habilitar la validación de esquemas en una colección mediante el uso del comando createCollection**. Sintaxis:

```
db.createCollection("miColeccion", {
  validator: {
    $jsonSchema: {
      // Aquí va el JSON Schema para la validación
    }
  }
})
```

Después de habilitar la validación de esquemas, *MongoDB* **se asegurará de que cada documento insertado o actualizado en la colección cumpla con el esquema especificado**. Si un documento no cumple con el esquema, se producirá un error y el documento no se insertará o actualizará.

También **es posible agregar un validador de esquemas a una colección existente utilizando el método db.createIndex() con la opción validator**.

Es importante tener en cuenta que **la validación de esquemas no es adecuada para todas las situaciones y puede afectar el rendimiento de las consultas en grandes conjuntos de datos**. Por lo tanto, es recomendable evaluar cuidadosamente si la validación de esquemas es necesaria para su caso de uso específico.

Ejemplo: crear una colección llamada personas con un esquema de validación.

```
db.createCollection("personas", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["nombre", "email"],
      properties: {
        nombre: {
          bsonType: "string",
          description: "Nombre de la persona"
        },
        email: {
          bsonType: "string",
          pattern: "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$",
          description: "Dirección de correo electrónico de la persona"
        },
        edad: {
          bsonType: "int",
          minimum: 0,
          maximum: 120,
          description: "Edad de la persona"
        },
        estado: {
          bsonType: "string",
          description: "Toma uno de los siguientes valores: activo, inactivo, suspendido",
          enum: [ "activo", "inactivo", "suspendido" ]
        },
        sexo: {
          type: "string",
          enum: ["masculino", "femenino"]
        }
      }
    }
  }
})
```

```

    }
  }
}
})

```

En este ejemplo, el esquema de validación especifica que los documentos en la colección `personas` deben tener un campo `nombre` de tipo `string` y un campo `email` de tipo `string` que cumpla con el patrón de una dirección de correo electrónico. Además, los campos `edad`, `estado` y `sexo` son opcionales, pero si se incluyen, la `edad` debe ser un entero entre 0 y 120, y los campos `estado` y `sexo` deben tomar uno de los valores indicados.

Para insertar un documento que cumpla con el esquema de validación anterior:

```

db.personas.insertOne({
  nombre: "Juan",
  email: "juan@ieshlanz.es",
  edad: 30,
  estado: "activo"
})

```

Si se intenta insertar un documento que no cumpla con el esquema de validación, *MongoDB* generará un error y no permitirá la inserción del documento en la colección:

```

db.personas.insertOne({
  nombre: "Pepe",
  email: "pepe@ieshlanz.es",
  edad: 130,
  estado: "activo"
})

```

```

> db.personas.insertOne({
  nombre: "Pepe",
  email: "pepe@ieshlanz.es",
  edad: 130,
  estado: "activo"
})
MongoServerError: Document failed validation

```

Más información en <https://www.mongodb.com/docs/manual/core/schema-validation/#schema-validation>

3.19 Usuarios.

En *MongoDB*, los usuarios se utilizan para autenticar el acceso a la *BD* y para controlar los permisos de acceso a las *BD* y colecciones individuales.

En los siguientes puntos se verán en más detalle algunos de ellos.

En la imagen se pueden ver todos los métodos disponibles para trabajar con usuarios.

Name	Description
<code>db.auth()</code>	Authenticates a user to a database.
<code>db.changeUserPassword()</code>	Changes an existing user's password.
<code>db.createUser()</code>	Creates a new user.
<code>db.dropUser()</code>	Removes a single user.
<code>db.dropAllUsers()</code>	Deletes all users associated with a database.
<code>db.getUser()</code>	Returns information about the specified user.
<code>db.getUsers()</code>	Returns information about all users associated with a database.
<code>db.grantRolesToUser()</code>	Grants a role and its privileges to a user.
<code>db.removeUser()</code>	Deprecated. Removes a user from a database.
<code>db.revokeRolesFromUser()</code>	Removes a role from a user.
<code>db.updateUser()</code>	Updates user data.
<code>passwordPrompt()</code>	Prompts for the password as an alternative to specifying passwords directly in various <i>mongosh</i> user authentication/management methods.

Más información en <https://www.mongodb.com/docs/manual/reference/method/js-user-management/>

3.19.1 Activar la autenticación y agregar el usuario root a MongoDB en Windows.

Ejecutar los comandos:

```
use admin
db.createUser({user: 'root', pwd: 'root', roles: [{role: 'root', db: 'admin'}]})
```

Ir al directorio de instalación de *MongoDB* y acceder a la carpeta *bin*, en esta carpeta editar el archivo *mongod.cfg* y en el apartado *#security* añadir las siguientes líneas (se debe realizar con permisos de *Administrador*):

```
#security:
security:
  authorization: enabled
```

Si se desea permitir el acceso remoto desde otros equipos al servidor de *MongoDB*, editar el archivo *mongod.cfg* y en el apartado *#network* interfaces comentar (haciendo uso de *#*) la línea *bindIp*.

Una vez realizados los cambios reiniciar el servicio de *MongoDB* para que los cambios surtan efecto y de esta forma tener activada la autenticación y el acceso remoto.

3.19.2 Crear un usuario.

Los usuarios se crean para cada una de las *BD*. Para crear un nuevo usuario se puede utilizar el comando *db.createUser()*. Su sintaxis es:

```
db.createUser({ user: "<nombre_de_usuario>",
                pwd: "<contraseña>",
                roles: [{role: "<rol>", db: "<base_de_datos>"},
                      {role: "<rol>", db: "<base_de_datos>"},
                      ... ],
                authenticationRestrictions: [{clientSource: ["<dirección_IP>"],
                                                serverAddress: ["<dirección_IP>"]},
                                             ... ],
                customData: {<campo>: <valor>, ...}
                })
```

, donde:

- *<nombre_de_usuario>*: es el nombre de usuario que se está creando.
- *<contraseña>*: es la contraseña asociada al usuario.
- *<rol>*: es el rol que se está asignando al usuario.
- *<base_de_datos>*: es la *BD* en la que se está asignando el rol.
- *<dirección_IP>*: es la dirección *IP* de la que se permite la autenticación. Si no se especifica, se permitirá la autenticación desde cualquier dirección *IP*.
- *<campo>* y *<valor>*: son campos y valores personalizados que se pueden agregar a la información del usuario.

Los roles disponibles son: *read*, *readWrite*, *dbAdmin*, *userAdmin*, *dbOwner*, *clusterAdmin*, *backup*, *restore*, *readAnyDatabase*, *readWriteAnyDatabase*, *userAdminAnyDatabase*, *dbAdminAnyDatabase*, *root*, ...

La opción *authenticationRestrictions* permite restringir la autenticación a direcciones *IP* específicas, mientras que la opción *customData* permite agregar información personalizada al documento del usuario.

Es importante tener en cuenta que el comando *db.createUser()* solo se puede ejecutar en una *BD* que ya existe y requiere permisos de administración.

Ejemplo: crear un usuario llamado *pepe* con contraseña *pepe* y permisos de lectura y escritura en la *BD* *conferencias* y permiso de lectura en la *BD* *conferencias2*.

```
db.createUser({user: "pepe", pwd: "pepe", roles: [{role: "readWrite", db: "conferencias"},
                                                {role: "read", db: "conferencias2"}]})
```

Más información en <https://www.mongodb.com/docs/manual/reference/method/db.createUser/>

3.19.3 Cambiar la contraseña de un usuario.

Para cambiar la contraseña de un usuario existente en una *BD* concreta, puede utilizarse el comando

`db.changeUserPassword()`. Su **sintaxis** es:

```
db.changeUserPassword(<username>, <password>)
```

, donde:

- **<username>**: es el nombre de usuario de la cuenta cuya contraseña se desea cambiar.
- **<password>**: es la nueva contraseña que se desea establecer para la cuenta.

Ejemplo: cambiar la contraseña de un usuario específico que exista en la *BD* activa.

```
db.changeUserPassword("usuario", "clave")
```

Más información en <https://www.mongodb.com/docs/manual/reference/method/db.changeUserPassword/>

3.19.4 Eliminar un usuario.

Para eliminar un usuario existente, puede utilizarse el **comando** `db.dropUser()`. Su **sintaxis** es:

```
db.dropUser(<username> [, <opciones>])
```

, donde:

- **<username>**: es el nombre de usuario del usuario que se eliminará.
- **<opciones>**: es un objeto que puede contener opciones adicionales. Es opcional.

Para **eliminar todos los usuarios** se puede utilizar el comando: `db.dropAllUsers([<opciones>])`

Es importante tener en cuenta que **la eliminación de un usuario no se puede deshacer** y, una vez eliminado, se deben actualizar todas las referencias a ese usuario en la aplicación y en la configuración de seguridad.

Ejemplo: eliminar un usuario sin opciones adicionales.

```
db.dropUser("pepe")
```

Ejemplo: eliminar un usuario con opciones adicionales.

```
db.dropUser("pepe", {writeConcern: {w: "majority", wtimeout: 5000}})
```

En este ejemplo, se especifica un objeto de opciones con una *writeConcern* que indica que la operación debe ser confirmada por la mayoría de los nodos en un conjunto de réplicas antes de que se considere exitosa, y que si la confirmación no se obtiene en 5 segundos, se debe producir un error de tiempo de espera.

Más información en <https://www.mongodb.com/docs/manual/reference/method/db.dropUser/>

3.19.5 Listar todos los usuarios.

Para listar todos los usuarios de la *BD* seleccionada se puede utilizar el **comando** `db.getUsers()`. Su **sintaxis** es:

```
db.getUsers({showCredentials: <boolean>, showCustomData: <boolean>, filter: <document>})
```

, donde:

- **showCredentials**: es opcional y toma el valor por defecto `false`. Establecerlo en `true` para mostrar el hash de la contraseña del usuario.
- **showCustomData**: es opcional y toma el valor por defecto `true`. Establecerlo en `false` para omitir los datos personalizados.
- **filter**: es opcional. Es un documento que especifica las condiciones de la etapa `$match` para devolver información a los usuarios que coinciden con las condiciones del filtro.

Ejemplo: mostrar todos los usuarios registrados en la *BD*.

```
db.getUsers()
```

Ejemplo: obtener información sobre los usuarios.

```
db.getUsers({showCredentials: true})
```

Esto mostrará información detallada sobre los usuarios, incluyendo el nombre de usuario, la *BD* de autenticación, el rol y los permisos asignados, y otros detalles de autenticación.

Más información en <https://www.mongodb.com/docs/manual/reference/method/db.getUsers/>

3.19.6 Otorgar roles a un usuario.

Para otorgar permisos adicionales a un usuario existente se puede utilizar el **comando** `db.grantRolesToUser()`. Su **sintaxis** es:

```
db.grantRolesToUser("<username>",
    [{role: "<rolename>", db: "<database>"},
    {role: "<rolename>", db: "<database>"},
    ... ],
    { <writeConcern> }
)
```

, donde:

- **<username>**: es el nombre del usuario al que se le asignarán los roles.
- **[]**: es un arreglo que contiene los roles a asignar, cada rol debe especificarse como un objeto que contenga la propiedad `role` con el nombre del rol y la propiedad `db` con el nombre de la *BD* en la que se aplicará.
- **<writeConcern>**: es un documento que especifica la cantidad de nodos que deben confirmar la operación antes de que sea considerada como exitosa. Es opcional.

Tener en cuenta que, **para poder asignar roles a un usuario, se deben tener los permisos necesarios en la *BD* correspondiente**.

Ejemplo: asignar un rol a un usuario en una *BD* específica.

```
db.grantRolesToUser("pepe", [{role: "readWrite", db: "conferencias"}])
```

Ejemplo: asignar varios roles a un usuario en diferentes *BD*.

```
db.grantRolesToUser("pepe", [{role: "read", db: "conferencias"},
    {role: "dbAdmin", db: "jardineria"}])
```

Ejemplo: especificar el nivel de confirmación requerido para la operación.

```
db.grantRolesToUser("pepe", [{role: "readWrite", db: "conferencias2"}],
    {w: "majority", wtimeout: 5000})
```

Ejemplo: otorgar al usuario pepe permisos de lectura en la colección ponentes.

```
db.grantRolesToUser("pepe", [{role: "read", db: "conferencias", collection: "ponentes"}])
```

Más información en <https://www.mongodb.com/docs/manual/reference/method/db.grantRolesToUser/>

3.19.7 Revocar roles a un usuario.

Para revocar permisos a un usuario existente se puede utilizar `db.revokeRolesFromUser()`. Su **sintaxis** es:

```
db.revokeRolesFromUser("<username>", [<roles>], {<writeConcern>})
```

Ejemplo: revocar un rol a un usuario en una *BD* específica.

```
db.revokeRolesFromUser("pepe", [{role: "readWrite", db: "conferencias"}])
```

Más información en <https://www.mongodb.com/docs/manual/reference/method/db.revokeRolesFromUser/>

3.20 Transacciones.

MongoDB admite transacciones, lo que significa que puede agrupar varias operaciones juntas en una transacción atómica. Las transacciones **garantizan la integridad de los datos y aseguran que todas las operaciones se completen correctamente o que no se realice ninguna operación en absoluto**.

Las transacciones en *MongoDB* se pueden utilizar en aplicaciones que requieren una alta consistencia de datos,

como las aplicaciones financieras o de comercio electrónico. Las transacciones se pueden utilizar en una configuración de replicación o en un clúster de *sharding*.

Réplica: es una copia sincronizada de una BD. Las réplicas se utilizan para garantizar la disponibilidad y la redundancia de los datos en caso de fallo del servidor principal.

Fragmentación (sharding): es una técnica para dividir una BD en fragmentos (shards) distribuidos en varios servidores. La fragmentación se utiliza para escalar horizontalmente la BD y manejar grandes volúmenes de datos.

Las transacciones de *MongoDB* funcionan mediante la adopción de un modelo de bloqueo de dos fases. Durante la **primera fase, se adquieren los bloqueos necesarios** para todas las operaciones dentro de una transacción. En la **segunda fase, las operaciones dentro de la transacción se ejecutan y se confirman o se deshacen** según sea necesario.

Para utilizar transacciones en *MongoDB*, **se debe utilizar una versión que las admita**. A partir de la versión 4.0, *MongoDB* admite transacciones en una configuración de replicación y desde la **versión 4.2**, también en un clúster de *sharding*.

Lo primero que se tiene que hacer para utilizar las transacciones es conocer los métodos que ofrecen. Por un lado se tienen **métodos para gestionar la sesión**:

- **startSession()**: las transacciones están asociadas a una sesión, es por ello que siempre se necesitará abrir una sesión.
- **endSession()**: cuando se cierra la sesión, si hay alguna transacción ejecutándose, esta se abortará.

Y por otro los **métodos para controlar la transacción**:

- **Session.startTransaction()**: empieza la transacción multi-documento dentro de una sesión.
- **Session.commitTransaction()**: confirmar los cambios que se hayan realizado dentro de la transacción.
- **Session.abortTransaction()**: abortar la ejecución de la transacción y por lo tanto no grabar las modificaciones hechas a lo largo de la transacción.

Más información en <https://www.mongodb.com/docs/drivers/go/current/fundamentals/transactions/>

3.20.1 Ejemplo de transacción.

Ahora se va a ejecutar paso a paso una transacción desde la *shell* de *MongoDB* para ver cómo funcionan las transacciones.

Importante: tener *MongoDB* arrancado en formato **Replica Set**, ya que si se tiene en modo *standalone* no funcionarán las transacciones. La finalidad de un *Replica Set* es la de proporcionar una alta disponibilidad de las BD *MongoDB*. La idea consiste en tener corriendo varias instancias de *MongoDB* con el fin de que la información se replique entre ellas, de tal forma que si el nodo primario se cae, pueda ser remplazado automáticamente por otro.

Para probar el ejemplo de uso de transacciones lo más fácil es hacer uso de **MongoDB Atlas** en su capa gratuita. Otra opción es crear un *Replica Set* de *MongoDB* haciendo uso de *Docker*, los pasos se describen en el siguiente enlace <https://www.mongodb.com/compatibility/deploying-a-mongodb-cluster-with-docker>.

Lo primero será crear una BD con algo de contenido:

```
use transacciones
db.personas.drop()
db.createCollection("personas")
db.personas.insertOne({_id: 1, nombre: "Pepe", apellido: "Sánchez"})
db.personas.insertOne({_id: 2, nombre: "Juan", apellido: "Moreno"})
```

Ahora se creará la sesión y de la sesión se elegirá la colección sobre la que se va a actuar:

```
var sesion = db.getMongo().startSession()
var personas = sesion.getDatabase('transacciones').getCollection('personas')
```

Lo siguiente será crear la transacción mediante el método `startTransaction()`:

```
sesion.startTransaction({readConcern: {level: 'snapshot'}, writeConcern: {w: 'majority'}})
```

Y acto seguido se insertará un documento:

```
personas.insertOne({_id: 3, nombre: "Luis", apellido: "Ruiz"})
```

Al estar sin confirmar la transacción, si se hace una consulta sobre la *BD* de personas se verá lo siguiente:

```
personas.find()
{ "_id": 1, "nombre": "Pepe", "apellido": "Sánchez" }
{ "_id": 2, "nombre": "Juan", "apellido": "Moreno" }
```

Es decir, el documento todavía no se encuentra insertado en la colección. Pero sí en el ámbito de la transacción:

```
personas.find()
{ "_id": 1, "nombre": "Pepe", "apellido": "Sánchez" }
{ "_id": 2, "nombre": "Juan", "apellido": "Moreno" }
{ "_id": 3, "nombre": "Luis", "apellido": "Ruiz" }
```

Ahora lo que se hará será confirmar la transacción mediante `commitTransaction()` y cerrar la sesión:

```
sesion.commitTransaction()
sesion.endSession()
```

Ya se podrá comprobar que el documento insertado mediante la transacción se encuentra en la colección:

```
personas.find()
{ "_id": 1, "nombre": "Pepe", "apellido": "Sánchez" }
{ "_id": 2, "nombre": "Juan", "apellido": "Moreno" }
{ "_id": 3, "nombre": "Luis", "apellido": "Ruiz" }
```

Si se hubiera utilizado el método `abortTransaction()` no se habría visto reflejado.

• Modificaciones para arrays

- `$push`, añade un elemento a un array. Este ejemplo añade el tema MongoDB al libro con `db.libros.update({ codigo:1 }, { $push : {temas: "MongoDB" } })`
- `$addToSet`, agrega elementos a un array solo si estos no existen. En el ejemplo se añade el tema Base de datos a todos los libros que no lo tengan. Primero preguntamos si el libro tiene el campo temas. Para que se añada a todos los libros indicamos `multi:true`: `b.libros.update({ temas : { $exists:true } }, { $addToSet: {temas:"Base de datos" } }, {multi:true})`
- `$each`, se usa en conjunto con `$addToSet` o `$push` para indicar que se añaden varios elementos al array. `db.libros.update({codigo:1},{bush:{temas:{ $each:["JSON","XML"] } } }); db.libros.update({codigo:2},{ $addToSet :{temas: { $each: ["Eclipse","Developer"] } } })`
- `$pop`, elimina el primer o último valor de un array. Con valor `-1` borra el primero, con otro valor el último. En el ejemplo se borra el primer tema del libro con código 3: `db.libros.update({codigo:3},{ $pop: temas:-1 } })`
- `$pull`, elimina los valores de un array que cumplan con el filtro indicado. En el ejemplo se borran de todos los libros los elementos 'Base de datos' y 'JSON', si los tienen: `db.libros.update({}, { $pull:{ temas: { $in: ["Base de datos","JSON"] } } }, { multi: true })`



TAREA

1. Crear una *BD* en *MongoDB* con tu nombre y dentro de ella crear una colección llamada “juegos”. De los juegos se puede almacenar la siguiente información: título, género, precio y fecha de lanzamiento.
 - a) Insertar un juego rellenando todos los campos.
 - b) Insertar un juego sin indicar la fecha de lanzamiento.
 - c) Realizar una búsqueda de todos los juegos del mismo género y ordenarlos por título (ascendente).
2. Crear la colección “empleados” dentro de la *BD* “mibd” y añadir los siguientes documentos:
 - emp_no: 1, nombre: “Juan”, dep: 10, salario: 1000, fecha_alta: “10/10/1999”.
 - emp_no: 2, nombre: “Alicia”, dep: 10, salario: 1400, fecha_alta: “07/08/200”, oficio: “Profesora”.
 - emp_no: 3, nombre: “María Jesús”, dep: 20, salario: 1500, fecha_alta: “05/01/2005”, oficio: “Analista”, comision: 100.
 - emp_no: 4, nombre: “Alberto”, dep: 20, salario: 1100, fecha_alta: “15/11/2001”.
 - emp_no: 5, nombre: “Fernando”, dep: 30, salario: 1400, fecha_alta: “20/11/1999”, comision: 200, oficio: “Analista”.

Proponer las sentencias a ejecutar para realizar las siguientes operaciones:

 - a) Crear la *BD* y la colección.
 - b) Insertar los documentos dados.
 - c) Mostrar los empleados del departamento 10.
 - d) Mostrar los empleados del departamento 10 y 20.
 - e) Mostrar los empleados con salario > 1300 y oficio “Profesora”.
 - f) Subir el salario a todos los analistas 100€.
 - g) Decrementar la comisión en 20€ (-20€), sólo a los que tengan comisión.

3. Se precisa diseñar un blog de noticias donde los usuarios registrados puedan publicar sus comentarios:
- Cada usuario tiene un identificador único, un nombre, un seudónimo, una cuenta de *Twitter* y una descripción. Además, de forma opcional, los usuarios pueden proporcionar como datos su dirección postal (calle, número, puerta, código postal y ciudad) o sus teléfonos de contacto (pueden tener varios).
 - Las noticias tienen un identificador único, un título, un cuerpo y una fecha de publicación. Son publicadas por un usuario y pueden contener o no, una lista de tags.
 - Las noticias reciben comentarios, quedando registrado el usuario que lo escribió, el comentario escrito y el momento en el que lo hizo.

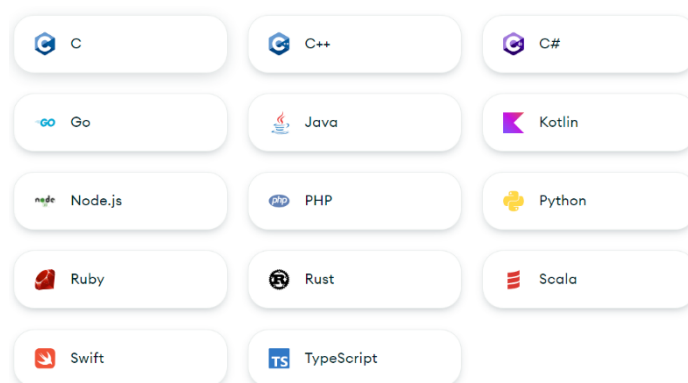
Se pide:

- Realizar el diagrama entidad/relación.
- Realizar el modelado de datos (definir las colecciones y como se van a establecer las relaciones).
- Insertar documentos de ejemplo en cada una de las colecciones.
- Mostrar los datos de un usuario dado su nombre de usuario y cuenta de *Twitter*.
- Mostrar el número de usuarios de cada código postal.
- Mostrar las 10 últimas noticias publicadas (más recientes).
- Mostrar el número de comentarios por noticia o por usuario (dos sentencias distintas).

4 Usando MongoDB y Java.

MongoDB ofrece gran cantidad de clientes para interactuar con él mediante gran cantidad de lenguajes de programación (<https://www.mongodb.com/docs/drivers/>).

En este punto se va a estudiar como trabajar con MongoDB mediante el driver Java para MongoDB.



4.1 Dependencias.

Para **agregar las librerías de MongoDB** para Java se puede seguir uno de los siguientes métodos:

- **Método 1.**
 - Descargar la librería de MongoDB desde <https://repo1.maven.org/maven2/org/mongodb/mongodb-driver-sync/5.3.0/mongodb-driver-sync-5.3.0.jar> y agregarla al proyecto (**Ant** o **IntelliJ**).
- **Método 2.** Usar Maven o Gradle.
 - **Maven:** agregar la siguiente dependencia en el archivo **pom.xml** dentro de la **sección <dependencies>**.

```
<dependencies>
  <dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongodb-driver-sync</artifactId>
    <version>5.3.0</version>
  </dependency>
</dependencies>
```

- **Gradle:** agregar la siguiente dependencia en el archivo **build.gradle** dentro de la **sección dependencies**.

```
dependencies {
    implementation 'org.mongodb:mongodb-driver-sync:5.3.0'
}
```

La elección del driver de MongoDB para Java depende de las necesidades específicas del proyecto:

- Driver síncrono: es ideal si se busca una implementación simple y secuencial, donde las operaciones se realicen de forma directa y sin complejidad adicional.
- Driver reactivo: si el proyecto requiere un alto rendimiento y escalabilidad, este driver es la mejor opción, ya que está diseñado para manejar múltiples operaciones de forma eficiente utilizando un enfoque no bloqueante.

Una vez configuradas las dependencias, hay que asegurarse de que estén disponibles para el proyecto, lo que puede requerir ejecutar el administrador de dependencias y actualizar el proyecto en el IDE.

4.2 Conexión a MongoDB.

Clases y métodos del controlador de MongoDB para Java a utilizar:

- ✓ **MongoClients:** es una **fábrica para crear instancias de MongoClient** (clase principal para interactuar con un

servidor *MongoDB*). Todos los **métodos** de la clase *MongoClients* **son estáticos**. Métodos más utilizados:

Método	Descripción
<code>MongoClient create()</code>	Crea un cliente predeterminado para conectarse al servidor MongoDB local.
<code>MongoClient create(String connectionString)</code>	Crea un cliente utilizando una cadena de conexión específica.
<code>MongoClient create(MongoClientSettings settings)</code>	Crea un cliente utilizando configuraciones avanzadas personalizadas.
<code>MongoClient create(ConnectionString connectionString)</code>	Crea un cliente usando un objeto <code>ConnectionString</code> que especifica la cadena de conexión.

Si no se pasan parámetros al método `create()`, se utiliza la configuración predeterminada del controlador de *MongoDB*: localhost, 27017 y sin usuario/clave (supone que el servidor *MongoDB* no tiene habilitada la autenticación).

El **formato de la URL de conexión** es el siguiente:

`mongodb[+srv]://<usuario>:<contraseña>@<host_clúster>:<puerto>/[<nombre_bd>][?opciones]`

, donde:

- `+srv`: se usa para la gestión de conexiones en entornos de clúster o replicación.
- `<usuario>`: nombre de usuario para la autenticación en la *BD*.
- `<contraseña>`: contraseña asociada con el usuario.
- `<host_clúster>`: dirección del servidor donde se ejecuta *MongoDB* (por ejemplo, localhost o una *IP*) o bien el nombre de dominio del clúster.
- `<puerto>`: puerto en el que *MongoDB* está escuchando (por defecto es 27017).
- `<nombre_bd>`: nombre de la *BD* a la que se desea conectarse (opcional; si no se especifica, se conecta a la *BD* predeterminada).
- `opciones`: parámetros adicionales opcionales como `authSource`, `replicaSet`, `ssl`, etc.

- ✓ **MongoClient**: representa una **conexión a un servidor MongoDB** y se utiliza para realizar operaciones en la *BD*. Métodos más utilizados:

Método	Descripción
<code>MongoIterable<String> listDatabaseNames()</code>	Devuelve un iterable con los nombres de las <i>BD</i> disponibles en el servidor.
<code>MongoIterable<Document> listDatabases()</code>	Devuelve un iterable con información detallada de las <i>BD</i> disponibles.
<code>MongoDatabase getDatabase(String databaseName)</code>	Devuelve un objeto <i>MongoDatabase</i> para interactuar con una <i>BD</i> específica.
<code>void close()</code>	Cierra la conexión del cliente con el servidor MongoDB.
<code>ClusterDescription getClusterDescription()</code>	Obtener la descripción del clúster del servidor MongoDB.

- ✓ **MongoDatabase**: representa una **BD en MongoDB** y se utiliza para realizar operaciones en esa *BD*. El método `listCollectionNames()` devuelve un iterable con los nombres de las colecciones de la *BD*. El resto de sus métodos se verán en un apartado posterior.

En *MongoDB*, no es necesario crear explícitamente una *BD* antes de comenzar a usarla. La *BD* se crea automáticamente cuando se inserta el primer documento en una colección.

Ejemplo: clase para manejar conexiones en *MongoDB*.

```
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
// import com.mongodb.ConnectionString;
```

```
public class MongoDBConnection {
    private MongoClient mongoClient; // Se inicializa automáticamente en null

    public MongoDBConnection(String user, String password, String host, int port) {
        String uri = String.format("mongodb://%s:%s@%s:%d", user, password, host, port);
        // ConnectionString cs = new ConnectionString(uri);
        try {
            if (mongoClient == null)
                mongoClient = MongoClient.create(uri); // mongoClient = MongoClient.create(cs);
            System.out.println("Conexión establecida.");
        } catch (Exception e) { // También se podría capturar MongoException
            System.err.println("Error al conectar con MongoDB: " + e.getMessage());
        }
    }
}
```

```
public MongoClient getConnection() { return mongoClient; }
```

```
public void closeConnection() {
    if (mongoClient != null) {
        try {
            mongoClient.close(); mongoClient = null;
            System.out.println("Conexión con MongoDB cerrada.");
        }
    }
}
```

<https://mongodb.github.io/mongo-java-driver/5.3/apidocs/mongodb-driver-sync/com/mongodb/client/package-summary.html>

URL conexión a un clúster de *MongoDB* con *Atlas*.

```
mongodb+srv://<username>:<password>@cluster0.7stmv.mongodb.net/myFirstDatabase?
retryWrites=true&w=majority
```

```

    } catch (Exception e) { // También se podría capturar MongoException
        System.err.println("Error al cerrar la conexión con MongoDB: " + e.getMessage());
    }
} else
    System.out.println("No hay conexión activa para cerrar.");
}
}

```

Ejemplo: mostrar información sobre el clúster de *MongoDB*, las *BD* disponibles en el servidor y las colecciones existentes en cada una de ellas.

```

import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.MongoIterable;
import com.mongodb.connection.ClusterDescription;
public class MostrarInfoServidorMongoDB {
    public static void main(String[] args) {
        MongoDBConnection dbConnection = new MongoDBConnection("root", "root", "localhost", 27017);
        MongoClient mongoClient = dbConnection.getConnection();
        if (mongoClient != null) {
            try {
                ClusterDescription clusterDescription = mongoClient.getClusterDescription();
                System.out.println("Descripción del clúster: " + clusterDescription.toString());
                System.out.println("BD y colecciones disponibles en el servidor MongoDB:");
                MongoIterable<String> databases = mongoClient.listDatabaseNames();
                for (String dbName : databases) {
                    System.out.println("\nBase de datos: " + dbName);
                    MongoDatabase database = mongoClient.getDatabase(dbName);
                    MongoIterable<String> collections = database.listCollectionNames();
                    System.out.println("  Colecciones:");
                    for (String collectionName : collections)
                        System.out.println("    - " + collectionName);
                }
            } catch (Exception e) {
                System.err.println("Error al obtener información: " + e.getMessage());
            }
        } else
            System.out.println("No hay conexión activa para obtener información.");
        dbConnection.closeConnection();
    }
}

```

Información sobre como instalar y configurar un logger

<https://www.mongodb.com/docs/drivers/java/sync/current/fundamentals/logging/>

4.3 Pool de conexiones.

Un pool de conexiones es una **técnica utilizada para administrar y reutilizar conexiones establecidas entre una aplicación y la *BD MongoDB***. En lugar de abrir y cerrar una conexión cada vez que se necesita interactuar con la *BD*, el pool de conexiones mantiene un conjunto de conexiones activas y disponibles para su reutilización.

Cuando una aplicación requiere una conexión a *MongoDB*, el pool de conexiones proporciona una conexión existente que no esté en uso. Después de que la aplicación haya terminado de usar la conexión, esta se devuelve al pool en lugar de cerrarse, lo que permite que se vuelva a utilizar para futuras solicitudes de conexión.

Si no se configura nada explícitamente, **por defecto se crea un pool de conexiones** con configuraciones predeterminadas. Esto sucede automáticamente **cuando se crea una instancia de *MongoClient***.

Los **valores por defecto** para el pool de conexiones son los siguientes:

- **maxSize:** 100 → número máximo de conexiones activas que puede tener el pool al mismo tiempo.
- **minSize:** 0 → número mínimo de conexiones que estarán abiertas en el pool simultáneamente.
- **maxWaitTime:** 120000 ms (120 segundos) → tiempo máximo que una solicitud de conexión esperará antes de lanzar una excepción si no hay conexiones disponibles.
- **maxConnectionLifeTime:** 0 (ilimitado) → tiempo máximo que una conexión puede permanecer en el pool

antes de ser cerrada.

- **maxConnectionIdleTime**: 0 (ilimitado) → tiempo máximo que una conexión puede permanecer inactiva antes de que se cierre.
- **maintenanceFrequency**: 60000 ms (60 segundos) → intervalo en el que el pool comprueba y elimina conexiones inactivas o vencidas.

Más información sobre otros parámetros de configuración en

<https://www.mongodb.com/docs/drivers/java/sync/current/fundamentals/connection/connection-options/>

El **pool de conexiones** se configura utilizando la clase **MongoClientSettings**. Aunque las configuraciones predeterminadas funcionan para muchas aplicaciones, se debería personalizar el pool de conexiones si:

- ✓ Se espera una alta concurrencia. Si se espera un alto número de operaciones simultáneas, ajustar **maxSize** para evitar cuellos de botella.
- ✓ Se requiere optimizar los recursos. Si el servidor tiene recursos limitados, ajustar **minSize** y **maxSize** para evitar una sobrecarga.
- ✓ Se necesitan bajas latencias. En aplicaciones críticas en tiempo de respuesta, reducir **maxWaitTime** para evitar retrasos innecesarios.
- ✓ Se quiere evitar tener conexiones inactivas. Ajustar **maxConnectionIdleTime** si se necesita liberar conexiones inactivas rápidamente.

Ejemplo: configuración del pool de conexiones.

```
import com.mongodb.ConnectionString;
import com.mongodb.MongoClientSettings;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import java.util.concurrent.TimeUnit;

public class MongoDBPoolEjemplo {
    public static void main(String[] args) {
        ConnectionString cs = new ConnectionString("mongodb://root:root@localhost:27017");
        // Configuración del cliente con pool de conexiones
        MongoClientSettings settings = MongoClientSettings.builder()
            // .applicationName("MongoDBPoolEjemplo")
            .applyConnectionString(cs)
            .applyToConnectionPoolSettings(builder -> builder
                .maxSize(50) // Tamaño máximo del pool
                .minSize(5) // Tamaño mínimo del pool
                .maxWaitTime(30, TimeUnit.SECONDS) // Tiempo máximo de espera
                .maxConnectionLifeTime(24, TimeUnit.HOURS)
                .maxConnectionIdleTime(30, TimeUnit.MINUTES))
            .build();
        try (MongoClient mongoClient = MongoClient.create(settings)) {
            System.out.println("Conexión a MongoDB establecida con pool de conexiones configurado.");
            var db = mongoClient.getDatabase("mi_base");
            System.out.println("Colecciones disponibles:"); // Listar las colecciones en la BD
            db.listCollectionNames().forEach(System.out::println);
            // Insertar un documento en una colección
            db.getCollection("prueba").insertOne(new org.bson.Document("clave", "valor"));
            System.out.println("Documento insertado en la colección 'prueba'.");
        }
    }
}
```

Ejemplo: valores por defecto del pool de conexiones creado por *MongoDB*.

```
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.MongoClientSettings;
import com.mongodb.connection.ConnectionPoolSettings;
import java.util.concurrent.TimeUnit;

public class MongoClientPoolDefaults {
    public static void main(String[] args) {
        MongoClientSettings settings = MongoClientSettings.builder().build();
        MongoClient mc = MongoClient.create(settings);
    }
}
```

```

    ConnectionPoolSettings ps = settings.getConnectionPoolSettings();
    System.out.println("Valores por defecto del pool de conexiones MongoDB:");
    System.out.println("maxSize: " + ps.getMaxSize());
    System.out.println("minSize: " + ps.getMinSize());
    System.out.println("maxConnectionLifeTimeMS: " + ps.getMaxConnectionLifeTime(TimeUnit.MILLISECONDS));
    System.out.println("maxConnectionIdleTimeMS: " + ps.getMaxConnectionIdleTime(TimeUnit.MILLISECONDS));
    System.out.println("maxWaitTimeMS: " + ps.getMaxWaitTime(TimeUnit.MILLISECONDS));
    System.out.println("maintenanceFrequency: " + ps.getMaintenanceFrequency(TimeUnit.MILLISECONDS));
}
}

```

4.4 Bases de datos, colecciones y documentos.

En *MongoDB*, los **datos se organizan en tres niveles**: *BD*, colecciones y documentos. Una **BD** es un **contenedor de colecciones** y se crea automáticamente al insertar el primer documento en ella. Las **colecciones son grupos de documentos**, similares a las tablas en *BD* relacionales. Un **documento es la unidad de almacenamiento y se representa en formato BSON**, que permite mayor flexibilidad que *JSON* e incluye tipos de datos adicionales.

La **clase *MongoDatabase*** proporciona **métodos** para trabajar con colecciones, algunos de los más utilizados son:

Método	Descripción
<code>MongoCollection<Document> getCollection(String collectionName)</code>	Devuelve un objeto <code>MongoCollection</code> para interactuar con una colección específica dentro de la BD.
<code>MongoIterable<String> listCollectionNames()</code>	Devuelve un iterable con los nombres de las colecciones de la BD.
<code>MongoIterable<Document> listCollections()</code>	Devuelve un iterable con información detallada de las colecciones de la BD.
<code>void createCollection(String collectionName)</code>	Crea una nueva colección en la BD.
<code>void drop()</code>	Elimina la BD actual.
<code>Document runCommand(Document command)</code>	Ejecuta un comando de BD, como <code>db.stats()</code> o <code>db.collection.drop()</code> .
<code>MongoCollection<T> getCollection(String collectionName, Class<T> clazz)</code>	Devuelve una colección con un tipo de objeto específico para trabajar con documentos de tipo <code>T</code> .
<code>MongoCollection<Document> getCollection(String collectionName, MongoCollection<Document> collection)</code>	Permite interactuar con una colección usando un tipo de documento más específico.

Tabla con los **métodos más utilizados** de la clase `MongoCollection`:

Método	Descripción
<code>long countDocuments()</code>	Devuelve el número de documentos que coinciden con un filtro opcional.
<code>void insertOne(T document)</code>	Inserta un solo documento en la colección.
<code>void insertMany(List<? extends T> documents)</code>	Inserta múltiples documentos en la colección.
<code>FindIterable<T> find()</code>	Devuelve un iterable para recorrer los documentos de la colección sin filtros.
<code>FindIterable<T> find(Bson filter)</code>	Devuelve un iterable para recorrer los documentos que coinciden con un filtro específico.
<code>void deleteOne(Bson filter)</code>	Elimina un solo documento que coincide con el filtro especificado.
<code>void deleteMany(Bson filter)</code>	Elimina todos los documentos que coinciden con el filtro especificado.
<code>UpdateResult updateOne(Bson filter, Bson update)</code>	Actualiza un solo documento que coincide con el filtro especificado.
<code>UpdateResult updateMany(Bson filter, Bson update)</code>	Actualiza todos los documentos que coinciden con el filtro especificado.
<code>void drop()</code>	Elimina la colección de la BD.

Tabla con los **métodos más utilizados** de la clase `org.bson.Document`:

Método	Descripción
<code>Object get(String key)</code>	Devuelve el valor asociado a la clave especificada.
<code>void put(String key, Object value)</code>	Inserta o actualiza el valor asociado a la clave especificada.
<code>boolean containsKey(String key)</code>	Comprueba si el documento contiene la clave especificada.
<code>Document append(String key, Object value)</code>	Agrega un par clave-valor al documento.
<code>Document putAll(Map<String, Object> map)</code>	Inserta múltiples pares clave-valor en el documento.
<code>Set<String> keySet()</code>	Devuelve un conjunto de todas las claves presentes en el documento.
<code>Document remove(String key)</code>	Elimina el par clave-valor asociado a la clave especificada.
<code>Object getObject(String key)</code>	Devuelve el valor asociado a la clave especificada, como un objeto.
<code>String toJson()</code>	Convierte el documento a su representación en formato <i>JSON</i> .
<code>Document parse(String json)</code>	Convierte una cadena <i>JSON</i> en un objeto <code>Document</code> .

4.5 Trabajar con bases de datos.

En *MongoDB*, **no se necesita un comando explícito para crear una BD**. Simplemente **seleccionar una BD y empezar a trabajar con ella**, creando colecciones e insertando documentos. **Esto automáticamente crea la BD**. La *BD* no existirá hasta que no se haya insertado al menos un documento en una colección dentro de ella o hasta que no se cree explícitamente una colección (aunque esta esté vacía).

Para **eliminar una BD completa**, usar el **método `dropDatabase`** de la clase `MongoDatabase`.

MongoDB no permite modificar propiedades de la BD, si permite gestionar colecciones o documentos.

Se puede usar el método `listDatabaseNames` de `MongoClient` para **obtener una lista de todas las BD en el servidor**.

Ejemplo: programa que combina la creación, listado y eliminación de varias BD.

```
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoDatabase;
import java.util.Scanner;

public class GestionBasesDeDatosMongoDB {
    public static void main(String[] args) {
        String uri = "mongodb://root:root@localhost:27017";
        try (MongoClient mongoClient = MongoClients.create(uri)) {
            Scanner scanner = new Scanner(System.in);
            int opcion;
            do {
                System.out.println("\n└─ Menú de gestión de BD ─┘");
                System.out.println("1. Crear una nueva BD.      ");
                System.out.println("2. Listar BD.                ");
                System.out.println("3. Eliminar una BD.         ");
                System.out.println("0. Salir.                   ");
                System.out.println("└──────────────────────────┘");
                System.out.print("Elige una opción: ");
                opcion = scanner.nextInt();
                scanner.nextLine(); // Limpiar el buffer de entrada
                switch (opcion) {
                    case 1: // Crear una nueva BD
                        System.out.print("Introduce el nombre de la nueva BD: ");
                        String nombreBaseDatosCrear = scanner.nextLine();
                        crearBaseDeDatos(mongoClient, nombreBaseDatosCrear);
                        break;
                    case 2: // Listar todas las BD
                        listarBasesDeDatos(mongoClient);
                        break;
                    case 3: // Eliminar una BD
                        System.out.print("Introduce el nombre de la BD a eliminar: ");
                        String nombreBaseEliminar = scanner.nextLine();
                        eliminarBaseDeDatos(mongoClient, nombreBaseEliminar);
                        break;
                    case 0:
                        System.out.println("Saliendo del programa.");
                        break;
                    default:
                        System.out.println("Opción no válida. Inténtalo nuevamente.");
                }
            } while (opcion != 0);
            scanner.close();
        } catch (Exception e) {
            System.err.println("Error en la conexión con MongoDB: " + e.getMessage());
        }
    }

    private static void crearBaseDeDatos(MongoClient mongoClient, String nombreBaseDatos) {
        try {
            MongoDatabase database = mongoClient.getDatabase(nombreBaseDatos);
            database.createCollection("micoleccion");
            System.out.println("La BD '" + nombreBaseDatos + "' ha sido creada con una colección.");
        } catch (Exception e) {
            System.err.println("Error al crear la BD: " + e.getMessage());
        }
    }

    private static void listarBasesDeDatos(MongoClient mongoClient) {
        try {
            System.out.println("\nBD disponibles en el servidor:");
            for (String nombreBaseDatos : mongoClient.listDatabaseNames())
                System.out.println("- " + nombreBaseDatos);
        } catch (Exception e) {
            System.err.println("Error al listar las BD: " + e.getMessage());
        }
    }
}
```

```

private static void eliminarBaseDeDatos(MongoClient mongoClient, String nombreBaseDatos) {
    try {
        MongoDBDatabase database = mongoClient.getDatabase(nombreBaseDatos);
        boolean baseDatosExiste = false;
        for (String dbName : mongoClient.listDatabaseNames()) {
            if (dbName.equals(nombreBase)) {
                baseDatosExiste = true;
                break;
            }
        }
        if (baseDatosExiste) {
            database.drop();
            System.out.println("La BD '" + nombreBaseDatos + "' ha sido eliminada.");
        } else {
            System.out.println("La BD '" + nombreBaseDatos + "' no existe.");
        }
    } catch (Exception e) {
        System.err.println("Error al eliminar la BD: " + e.getMessage());
    }
}
}

```

4.6 Trabajar con colecciones.

En *MongoDB*, las **colecciones** se crean automáticamente al insertar el primer documento, pero si se desea crearlas **explícitamente**, se puede usar el método `createCollection`.

Si se necesita configurar **parámetros específicos** para la **colección**, como el tamaño máximo o índices predeterminados, se puede usar `CreateCollectionOptions`:

```

CreateCollectionOptions options = new CreateCollectionOptions()
    .capped(true) // Activa el modo "capped" (tamaño fijo)
    .sizeInBytes(1024 * 1024) // Tamaño máximo en bytes
    .maxDocuments(100); // Máximo de documentos

database.createCollection("coleccionCapped", options);

```

Las **colecciones** **no se pueden modificar** directamente, las configuraciones se establecen en el momento de crearlas. Si se necesita cambiar estos parámetros, la única opción es eliminar la colección y recrearla con los ajustes deseados.

Para **eliminar colecciones** se puede usar el **método drop** de la clase `MongoCollection`. La operación drop elimina permanentemente la colección y todos los documentos que contiene (no hay forma de deshacer esta acción).

Para **listar las colecciones** existentes en una *BD* en *MongoDB* se puede usar el **método listCollectionNames** de la clase `MongoDatabase`.

Ejemplo: crear, listar y eliminar colecciones.

```

import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Collation;
import com.mongodb.client.model.CollationStrength;
import com.mongodb.client.model.CreateCollectionOptions;
import java.text.Collator;
import java.util.Locale;

public class GestionColecciones {
    public static void main(String[] args) {
        String uri = "mongodb://root:root@localhost:27017";
        try (MongoClient mongoClient = MongoClients.create(uri)) {
            MongoDBDatabase database = mongoClient.getDatabase("mi_bd");
            // Configuración del Collation (insensible a mayúsculas y acentos, idioma español)
            Collation collation = Collation.builder()
                .locale("es") // Idioma español
                .caseLevel(false) // Insensible a mayúsculas/minúsculas
                .collationStrength(CollationStrength.PRIMARY) // Ignorar acentos
                .build();
            // Configuración de opciones para la colección
            CreateCollectionOptions opciones = new CreateCollectionOptions()
                .capped(true) // Activar colección "capped" (tamaño limitado)

```

```

        .sizeInBytes(1024 * 1024) // Tamaño máximo en bytes (1 MB)
        .maxDocuments(100) // Máximo de documentos permitidos
        .collation(collation); // Configuración del Collation
String nombreColeccion = "mi_coleccion";
if (!existeColeccion(database, nombreColeccion)) { // Crear la colección si no existe
    database.createCollection(nombreColeccion, opciones);
    System.out.println("Colección '" + nombreColeccion + "' creada con parámetros.");
} else
    System.out.println("La colección '" + nombreColeccion + "' ya existe.");
listarColecciones(database); // Listar colecciones existentes
String coleccionAEliminar = "mi_coleccion"; // Eliminar una colección si existe
eliminarColeccion(database, coleccionAEliminar);
listarColecciones(database); // Listar nuevamente para comprobar la eliminación
} catch (Exception e) {
    System.err.println("Error: " + e.getMessage());
}
}

private static boolean existeColeccion(MongoDatabase database, String nombreColeccion) {
    for (String coleccion : database.listCollectionNames()) {
        if (coleccion.equals(nombreColeccion))
            return true;
    }
    return false;
}

private static void listarColecciones(MongoDatabase database) {
    System.out.println("\nColecciones en la BD '" + database.getName() + "':");
    for (String coleccion : database.listCollectionNames())
        System.out.println(" - " + coleccion);
}

private static void eliminarColeccion(MongoDatabase database, String nombreColeccion) {
    if (existeColeccion(database, nombreColeccion)) {
        database.getCollection(nombreColeccion).drop();
        System.out.println("Colección '" + nombreColeccion + "' eliminada.");
    } else
        System.out.println("La colección '" + nombreColeccion + "' no existe.");
}
}

```

4.7 Creación de documentos.

En la interacción con *MongoDB* usando el driver de *Java* se suele utilizar para casi todo la clase **Document** (**org.bson.Document**). Se utilizará esta clase no sólo para los documentos que se tengan que introducir, sino para definir los criterios de eliminación, actualización, ordenación, etc.

El constructor de la clase **Document** admite dos parámetros: una clave y un valor. Para crear documentos con más de una pareja clave-valor se aplicará el método **append()** al constructor tantas veces como sea necesario encadenando las llamadas con un punto. Esta será la técnica básica para construir las listas *JSON*.

```

ArrayList<String> misHobbies = new ArrayList<String>();
misHobbies.add("Música");
misHobbies.add("Cine");
Document documento = new Document("nombre", "Juan").
    append("email", "juan@gmail.com").
    append("twitter", "@juanmongodb").
    append("hobbies", misHobbies).
    append("localizacion", new Document("ciudad", "Granada").append("cp", 18001));

```

4.8 Inserción de documentos.

Los documentos se insertarán con los **métodos insertOne** o **insertMany** dependiendo de si se quiere **insertar uno o varios documentos** simultáneamente. Ambos métodos se aplican sobre la colección correspondiente.

Estos métodos de inserción de documentos permiten construir y almacenar datos en *MongoDB* de manera sencilla. Tener en cuenta que las excepciones **MongoWriteException** o **MongoException** pueden ser lanzadas durante las operaciones de inserción, así que es importante manejarlas adecuadamente en el código.

Ejemplo: insertar un documento complejo con un array en una clave y un documento anidado en otra.

```
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import java.util.ArrayList;
import org.bson.Document;

public class MongoDBInsertarDocumentoEjemplo {
    public static void main(String[] args) {
        String cs = "mongodb://root:root@localhost";
        try (MongoClient mongoClient = MongoClients.create(cs)) {
            MongoDatabase db = mongoClient.getDatabase("mibd");
            ArrayList<String> hobbies = new ArrayList<>();
            hobbies.add("Cine");
            hobbies.add("Deporte");
            hobbies.add("Informática");
            Document documento = new Document("nombre", "Juan Moreno").append("email", "juan@gmail.com")
                .append("twitter", "@juan").append("hobbies", hobbies).append("edad", 25)
                .append("localizacion", new Document("ciudad", "Granada").append("cp", "18001"));
            MongoCollection<Document> coleccion = db.getCollection("personas");
            coleccion.insertOne(documento); // Insertar el documento en la colección personas
        }
    }
}
```

```
{
  "_id": ObjectId('65d9e0f860629c370eb11e8f')
  nombre: "Juan Moreno"
  email: "juan@gmail.com"
  twitter: "@juan"
  hobbies: Array (3)
    0: "Cine"
    1: "Deporte"
    2: "Informática"
  edad: 25
  localizacion: Object
    ciudad: "Granada"
    cp: "18001"
}
```

Ejemplo: insertar una lista de documentos con insertMany().

```
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import java.util.ArrayList;
import org.bson.Document;

public class MongoDBInsertarVariosDocumentosEjemplo {
    public static void main(String[] args) {
        String cs = "mongodb://root:root@localhost";
        try (MongoClient mongoClient = MongoClients.create(cs)) {
            MongoDatabase db = mongoClient.getDatabase("mibd");
            ArrayList<Document> documentos = new ArrayList<>(); // Lista de documentos
            MongoCollection<Document> col = db.getCollection("contadores"); // Seleccionar la colección
            for (int i = 1; i <= 50; i++) { // Añadir nuevos documentos a la lista de documentos
                documentos.add(new Document("contador", i));
            }
            col.insertMany(documentos); // Insertar la lista de documentos en la colección
        }
    }
}
```

```
{
  "_id": ObjectId('679611f6f373633ce2e64b7f')
  contador: 1
}
{
  "_id": ObjectId('679611f6f373633ce2e64b80')
  contador: 2
}
{
  "_id": ObjectId('679611f6f373633ce2e64b81')
  contador: 3
}
{
  "_id": ObjectId('679611f6f373633ce2e64b82')
  contador: 4
}
{
  "_id": ObjectId('679611f6f373633ce2e64b83')
  contador: 5
}
{
  "_id": ObjectId('679611f6f373633ce2e64b84')
  contador: 6
}
```

4.9 Búsqueda de documentos.

La búsqueda de documentos **se realiza mediante el método find() aplicado a una colección**. El método find() devuelve un objeto de la clase FindIterable<Document>. Sobre un objeto FindIterable se puede usar el método first(), que como su nombre indica **devuelve el primer elemento** del objeto FindIterable.

Sintaxis básica del método find de MongoCollection:

- FindIterable<Document> find()
- FindIterable<Document> find(Document filter)
- FindIterable<Document> find(Document filter, Document projection)

Si se necesita buscar en varias colecciones, se tendrán que realizar operaciones de búsqueda por separado en cada colección. Si aun así se necesita realizar operaciones más complejas que involucren datos de varias colecciones, **se podría considerar el uso de la agregación en MongoDB** (ver punto 4.12), que permite combinar datos de múltiples colecciones en una operación.

Tabla con los **métodos más utilizados** de FindIterable<Document>:

Método	Descripción	Ejemplo de Uso
void forEach(Consumer<? super T> action)	Itera sobre todos los documentos en la colección.	find().forEach(doc -> System.out.println(doc.toJson()));

<code>FindIterable<Document> limit(int limit)</code>	Limita la cantidad de documentos devueltos por la consulta.	<code>find().limit(5)</code> → Devuelve solo los primeros 5 documentos.
<code>FindIterable<Document> skip(int skip)</code>	Omite una cantidad determinada de documentos al principio de los resultados.	<code>find().skip(5)</code> → Omite los primeros 5 documentos.
<code>FindIterable<Document> sort(Bson sort)</code>	Ordena los resultados según los criterios especificados.	<code>find().sort(new Document("edad", 1))</code> → Ordena los documentos por el campo edad en orden ascendente.
<code>FindIterable<Document> projection(Bson projection)</code>	Limita los campos devueltos por cada documento (proyección).	<code>find().projection(new Document("nombre", 1).append("edad", 1))</code> → Solo devuelve los campos nombre y edad.
<code>Document first()</code>	Devuelve el primer documento que coincide con la consulta.	<code>Document doc = find().first();</code> → Obtiene el primer documento que coincida con la consulta.
<code>FindIterable<Document> collation(Collation collation)</code>	Establece la configuración de collation para la consulta (por ejemplo, insensible a mayúsculas/minúsculas).	<code>find().collation(Collation.builder().locale("es").caseLevel(false).strength(Collation.ComparisonLevel.PRIMARY).build())</code>
<code>Iterator<Document> iterator()</code>	Devuelve un iterador para recorrer los resultados de la consulta.	<code>Iterator<Document> iter = find().iterator();</code> <code>while (iter.hasNext()) {</code> <code>System.out.println(iter.next().toJson());</code> <code>}</code>

Ejemplo: aplicar un filtro por edad, omitir los primeros 5 documentos, limitar los resultados a 10, ordenar por el campo edad y mostrar solo los campos nombre y edad.

```
collection.find(new Document("edad", new Document("$gte", 18))) // Aplica el filtro
.skip(5) // Omite los primeros 5 documentos
.limit(10) // Limita a 10 documentos
.sort(new Document("edad", 1)) // Ordena por edad en orden ascendente
.projection(new Document("nombre", 1).append("edad", 1)) // Solo muestra nombre y edad
.forEach(doc -> System.out.println(doc.toJson()));
```

Ejemplo: resultado de una consulta en forma de documento en *JSON* - método `toJson()`.

```
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import org.bson.Document;
import java.util.Objects;

public class MongoDBBuscarDocumentoEjemplo {
    public static void main(String[] args) {
        String cs = "mongodb://root:root@localhost";

        try (MongoClient mongoClient = MongoClients.create(cs)) {
            MongoDatabase db = mongoClient.getDatabase("mibd");
            MongoCollection<Document> col = db.getCollection("personas");

            // Crear el documento para especificar el criterio de búsqueda
            Document criterio = new Document("email", "juan@gmail.com");

            // Documento para almacenar los resultados
            FindIterable<Document> resultadoDocumento = col.find(criterio);

            // Muestra el primer documento encontrado
            System.out.println("Documento: " + resultadoDocumento.first());

        }
    }
}
```

Ejemplo con más de una condición que devuelve más de un resultado.

Si en lugar de recuperar el primer documento, se quiere **recorrer el conjunto de resultados**, se tiene que **convertir el objeto FindIterable que devuelve el método find() en un cursor**. Para ello se usará el **método iterator()**. El método `iterator()` devuelve un objeto `MongoCursor` que se podrá recorrer con el método `next()`.

`MongoCursor` es una interfaz de la API de *MongoDB* para Java. Se utiliza para **iterar sobre los documentos devueltos por una consulta en una colección de MongoDB**. Es similar a un iterador en Java, pero está específicamente diseñado para manejar resultados provenientes de una *BD MongoDB*.

```
try (MongoClient mc = MongoClients.create("mongodb://root:root@localhost")) {
    MongoDatabase db = mc.getDatabase("conferencias");
    MongoCollection<Document> col = db.getCollection("conferencias");

    // Construir la consulta utilizando operadores de agregación
    Document criterio = new Document("$or", Arrays.asList(
        new Document("precio", new Document("$gte", 15.5))
            .append("turno", "T"),
        new Document("precio", new Document("$lte", 15.5))
            .append("turno", "M")
    ));

    // Aplicar la consulta
    FindIterable<Document> iterable = col.find(criterio);

    // Iterar sobre los resultados
    System.out.println(x:"Conferencias que cumplen las condiciones:");
    for (Document doc : iterable) {
        System.out.println(" - " + doc.getString("tema") + " (" +
            doc.getDouble("precio") + "€ - " + doc.getString("turno") + ")");
    }
}
```

Tabla con los **métodos más utilizados** de `MongoCursor`:

Método	Descripción
<code>boolean hasNext()</code>	Devuelve true si hay más documentos en el cursor.
<code>Document next()</code>	Devuelve el siguiente documento del cursor. Lanza una excepción si no hay más elementos.
<code>Document tryNext()</code>	Devuelve el siguiente documento o null si no hay más elementos.
<code>void close()</code>	Cierra el cursor y libera los recursos asociados.

Cuando se tiene un documento de *MongoDB* en un objeto de tipo **Document**, se puede **acceder a los valores de sus campos** utilizando diferentes métodos según el tipo de dato que se espera (similares a los usados con *JDBC*):

Método	Descripción
<code>Object get(String key)</code>	Devuelve el valor del campo como un objeto genérico (Object).
<code>String getString(String key)</code>	Devuelve el valor del campo como una cadena de texto (String).
<code>Integer getInteger(String key)</code>	Devuelve el valor del campo como un entero (Integer).
<code>Double getDouble(String key)</code>	Devuelve el valor del campo como un número decimal (Double).
<code>Boolean getBoolean(String key)</code>	Devuelve el valor del campo como un valor booleano (Boolean).
<code>Date getDate(String key)</code>	Devuelve el valor del campo como una fecha (java.util.Date).
<code>List<T> getList(String key, Class<T>)</code>	Devuelve el valor del campo como una lista (List<T>).
<code>T getEmbedded(List<String>, Class<T>)</code>	Devuelve un valor anidado dentro de un subdocumento especificando su ruta.
<code>String toJson()</code>	Devuelve el documento completo como una cadena en formato JSON.

Ejemplo:

```
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.MongoCursor;
import java.util.List;
import org.bson.Document;

public class MongoDBCollectionEjemplo {
    public static void main(String[] args) {
        String connectionString = "mongodb://root:root@localhost:27017";
        String nombreBD = "mibd", nombreColeccion = "amigos";

        try (MongoClient clienteMongo = MongoClient.create(connectionString)) {
            MongoDatabase bd = clienteMongo.getDatabase(nombreBD);

            bd.createCollection(nombreColeccion); // Crear una nueva colección
            System.out.println("Colección creada: " + nombreColeccion);

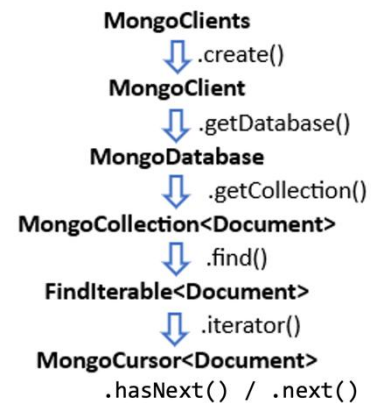
            // Insertar documentos en la colección
            MongoCollection<Document> coleccion = bd.getCollection(nombreColeccion);
            Document amigo1 = new Document("nombre", "Juan").append("edad", 30).append("ciudad", "Granada");
            Document amigo2 = new Document("nombre", "Alicia").append("edad", 25).append("ciudad", "Málaga");
            coleccion.insertMany(List.of(amigo1, amigo2));
            System.out.println("Amigos insertados en la colección.");

            // Consultar todos los documentos en la colección
            System.out.println("Documentos de amigos en la colección:");
            coleccion.find().forEach(doc -> System.out.println(doc.toJson()));

            // Leer y procesar los documentos con métodos como getString y getInteger
            System.out.println("\nProcesando documentos de amigos:");
            try (MongoCursor<Document> cursor = coleccion.find().iterator()) {
                while (cursor.hasNext()) {
                    Document doc = cursor.next();
                    System.out.print(" - Nombre: " + doc.getString("nombre"));
                    System.out.print(", edad: " + doc.getInteger("edad"));
                    System.out.println(", ciudad: " + doc.getString("ciudad"));
                }
            }

            // Buscar un documento específico
            Document resultado = coleccion.find(new Document("nombre", "Juan")).first();
            if (resultado != null) {
                System.out.println("\nDocumento encontrado:");
                System.out.println(" Nombre: " + resultado.getString("nombre"));
                System.out.println(" Edad: " + resultado.getInteger("edad"));
                System.out.println(" Ciudad: " + resultado.getString("ciudad"));
            } else {
                System.out.println("\nNo se encontró el documento especificado.");
            }

            bd.getCollection(nombreColeccion).drop(); // Eliminar la colección
            System.out.println("\nColección eliminada: " + nombreColeccion);
        }
    }
}
```



Ejemplo: *MongoDB* no proporciona un comando para renombrar *BD*, pero se puede crear la nueva *BD* con el nombre deseado y copiar los datos desde la *BD* original a la nueva.

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

public class RenombrarBD {
    public static void renombrarBD(MongoClient cliente, String bdOrigen, String bdDestino) {
        MongoDatabase origen = cliente.getDatabase(bdOrigen);
        MongoDatabase destino = cliente.getDatabase(bdDestino);

        for (String nombreColeccion : origen.listCollectionNames()) { // Lista de colecciones en la BD origen
            // Obtener la colección actual en la BD origen
            MongoCollection<Document> coleccionOrigen = origen.getCollection(nombreColeccion);
            // Obtener o crear la colección correspondiente en la BD de destino
            MongoCollection<Document> coleccionDestino = destino.getCollection(nombreColeccion);
            // Copiar los documentos de la colección origen a la colección de destino
            coleccionOrigen.find().forEach(documento -> coleccionDestino.insertOne(documento));
            System.out.println("Datos copiados de " + nombreColeccion + " a la nueva BD: " + bdDestino);
        }
        System.out.println("Todos los datos se han copiado correctamente.");
    }

    public static void main(String[] args) {
        String nombreBD = "mibd", nuevoNombreBD = "agenda", nombreColeccion = "contactos";
        MongoClient clienteMongo = null;
        try {
            // Conectar y seleccionar la BD usando la clase MongoDBConnection
            MongoDBConnection cm = new MongoDBConnection("root", "root", "localhost", 27017);
            clienteMongo = cm.getConnection(); // Obtener el cliente MongoDB
            MongoDatabase baseDatos = clienteMongo.getDatabase(nombreBD);
            System.out.println("Conexión realizada correctamente a la BD.");

            // Insertar un documento en la colección para crear la BD
            MongoCollection<Document> coleccion = baseDatos.getCollection(nombreColeccion);
            Document doc = new Document("nombre", "Juan");
            coleccion.insertOne(doc);
            System.out.println("Documento insertado en la colección " + nombreColeccion);

            // Copiar datos desde la BD original a la nueva
            renombrarBD(clienteMongo, nombreBD, nuevoNombreBD);

            baseDatos.drop(); // Eliminar la BD original si es necesario
            System.out.println("BD eliminada: " + baseDatos.getName());

            cm.closeConnection(); // Desconectar
        } catch (Exception e) {
            System.err.println("Error: " + e.getMessage());
        }
    }
}
```

```
// Con el logger slf4j-simple --> eliminar salidas de log
Logger.getLogger( name: "org.mongodb.driver").setLevel(Level.SEVERE);
MongoClient mc = MongoClient.create("mongodb://root:root@localhost");
```

```
// Aplicar la consulta
FindIterable<Document> iterable = col.find(
    Filters.or(
        Filters.and(
            Filters.gte("precio", 15.5),
            Filters.eq("turno", "T")
        ),
        Filters.and(
            Filters.lte("precio", 15.5),
            Filters.eq("turno", "M")
        )
    )
);
```

Ejemplo con más de una condición.

4.9.1 Filters.

Filters es una **clase** que **permite construir filtros para consultas**. Esta clase se utiliza para construir expresiones de filtrado que especifican condiciones para seleccionar documentos en una colección.

Tabla con algunos **ejemplos comunes** de la clase **Filters**:

Operación	Ejemplo de filtro
Igualdad	<code>Filters.eq("nombre", "Juan Moreno")</code>
Mayor que	<code>Filters.gt("edad", 25)</code>
Menor que	<code>Filters.lt("edad", 30)</code>
Mayor o igual que	<code>Filters.gte("edad", 25)</code>
Menor o igual que	<code>Filters.lte("edad", 30)</code>
Combinar con and	<code>Filters.and(Filters.eq("campo1", valor1), Filters.eq("campo2", valor2))</code>
Combinar con or	<code>Filters.or(Filters.eq("campo1", valor1), Filters.eq("campo2", valor2))</code>
Expresiones regulares	<code>Filters.regex("nombre", "^J")</code>

Recordar que estas expresiones de filtro se utilizan en combinación con métodos como `find()`, `deleteOne()`, `deleteMany()`, `updateOne()` y similares.

Ejemplo:

```
FindIterable<Document> resultado = collection.find(
    Filters.and(
        Filters.gte("edad", 25), // Edad mayor o igual a 25
        Filters.regex("nombre", "^J") // Nombre que comienza con "J"
    ));
```

4.9.2 Ordenación, proyección, skip y limit.

Además de `first()`, la clase **FindIterable** proporciona **otros métodos** interesantes:

- ✓ **sort()**: toma un documento como argumento indicando los campos por los que se quiere ordenar y el criterio de ordenación: ascendente (1) o descendente (-1), de la misma forma que en la *shell* de *MongoDB*.
- ✓ **projection()**: toma un documento como argumento indicando los campos que se quieren mostrar (1) u ocultar (0), de la misma forma que en la *shell* de *MongoDB*.
- ✓ **skip()** y **limit()**: toman un entero como argumento que representa el número de documentos a saltar y limitar, respectivamente.

Nota: `sort()`, `projection()`, `skip()` y `limit()` devuelven un objeto `FindIterable`. Para poder recorrerlo hay que convertirlo en un cursor aplicando el método `iterator()`.

Ejemplo: método que muestra las personas que se encuentran en una localización dada.

```
public void encontrarPersonasProyectarOrdenar(MongoDatabase db, String localizacion) {
    // Seleccionar la colección personas
    MongoCollection<Document> coleccion = db.getCollection("personas");
    // Crear el documento con el criterio de búsqueda
    Document documentoBusqueda = new Document("localizacion.ciudad", localizacion);
    // Crear el documento con el criterio de ordenación
    Document documentoOrdenacion = new Document("nombre", 1);
    // Crear el documento para especificar los criterios de proyección
    Document documentoProyeccion = new Document("localizacion", 1).append("nombre", 1);
    // Documento para almacenar los resultados
    MongoCursor<Document> documentoResultado = coleccion.find(documentoBusqueda)
        .sort(documentoOrdenacion).projection(documentoProyeccion).limit(5).iterator();
    System.out.println("*** Listado de 5 personas que viven en " + localizacion);
    while (documentoResultado.hasNext()) {
        Document documento = documentoResultado.next();
        System.out.println(documento.getString("nombre") + " " +
            ((Document) documento.get("localizacion")).getString("cp"));
        // "localizacion.cp" no puede ser directamente accedido
    }
}
```

Ejemplo: inserción de documentos y uso de `find()` en combinación con `sort()`, `projection()`, `skip()` y `limit()`.

```
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Filters;
import com.mongodb.client.model.Projections;
import com.mongodb.client.model.Sorts;
import org.bson.Document;
import java.util.Arrays;

public class MongoDBEjemploSortProjectSkipLimit {
    public static void main(String[] args) {
        String cs = "mongodb://root:root@localhost";
        try (MongoClient mongoClient = MongoClients.create(cs)) {
            MongoDatabase database = mongoClient.getDatabase("mibd");
            MongoCollection<Document> personasColeccion = database.getCollection("personas2");
            // Insertar solo si la colección está vacía (para evitar duplicados en este ejemplo)
            if (personasColeccion.countDocuments() == 0) {
                // Crear documentos Persona
                Document persona1 = new Document("nombre", "Juan").append("edad", 30);
                Document persona2 = new Document("nombre", "María").append("edad", 25);
                Document persona3 = new Document("nombre", "Luis").append("edad", 28);
                Document persona4 = new Document("nombre", "Francisco").append("edad", 39);
                Document persona5 = new Document("nombre", "Pepa").append("edad", 52);

                // Insertar en la colección
                personasColeccion.insertMany(Arrays.asList(persona1, persona2, persona3, persona4, persona5));
                System.out.println("Personas insertadas en la colección.");
            }
        }
    }
}
```

```

    }
    // Búsqueda de personas en la colección con filtrado, ordenamiento, proyección, skip y limit
    System.out.println("Personas en la colección:");
    personasColeccion
        .find(Filters.gte("edad", 28))           // Filtrar por edad mayor o igual a 28
        .projection(Projections.exclude("_id")) // Excluir el campo _id de la salida
        .sort(Sorts.ascending("nombre"))        // Ordenar por nombre ascendente
        .skip(1)                                // Saltar el primer resultado
        .limit(2)                                // Limitar a 2 resultados
        .forEach(document -> System.out.println(document.toJson()));
    }
}

```

4.9.3 Colaciones.

collation es una característica de *MongoDB* que **define cómo se comparan y ordenan las cadenas de texto**. Esto incluye aspectos como la sensibilidad a mayúsculas y minúsculas, el manejo de acentos y la configuración del idioma. Es útil para realizar búsquedas, ordenaciones y comparaciones personalizadas en las colecciones de *MongoDB*.

Parámetros importantes del collation:

- **locale:** especifica el idioma o configuración regional (por ejemplo, "en" para inglés, "es" para español, "fr" para francés, etc.).
- **strength:** determina la sensibilidad de la comparación.
 - 1 (primary): insensible a mayúsculas, acentos, y otros.
 - 2 (secondary): insensible a mayúsculas, pero sensible a acentos.
 - 3 (tertiary): sensible a mayúsculas y acentos.
- **caseLevel:** si se establece en true, considera la sensibilidad a mayúsculas cuando strength es 1 o 2.
- **numericOrdering:** si está habilitado (true), los números se ordenan por su valor (por ejemplo, 2 antes de 10).
- **alternate:** controla cómo se tratan los caracteres no distinguibles, como los espacios.

Se puede establecer un collation predeterminado en el momento de crear una colección:

```
db.createCollection("miColeccion", {collation: {locale: "es", strength: 2}});
```

Se puede aplicar un collation directamente en las operaciones de consulta, incluso si no está definido en la colección:

```
db.miColeccion.find({nombre: "café"}, {collation: {locale: "es", strength: 1}});
```

El **uso de collation requiere índices compatibles**. Si se van a realizar búsquedas con un collation específico, asegurarse de que los índices también lo utilicen, para evitar degradaciones de rendimiento:

```
db.miColeccion.createIndex({nombre: 1}, {collation: {locale: "es", strength: 2}});
```

En *Java*, al trabajar con el driver oficial de *MongoDB*, la **clase Collation** permite configurar opciones de collation. Esto se utiliza para consultas, índices y operaciones en *MongoDB* que necesitan un comportamiento personalizado en la comparación y ordenamiento de cadenas.

Nivel (strength)	Constante (CollationStrength)	Sensibilidad	"café" vs. "CAFE"	"cafe" vs. "CAFE"
1 (Primary)	CollationStrength.PRIMARY	Solo distingue caracteres base (ignora mayúsculas y acentos).	Iguales	Iguales
2 (Secondary)	CollationStrength.SECONDARY	Distingue acentos, pero ignora mayúsculas.	Diferentes	Iguales
3 (Tertiary)	CollationStrength.TERTIARY	Distingue mayúsculas y acentos.	Diferentes	Diferentes

Ejemplo: configurar el collation al realizar consultas con el driver de *MongoDB*.

```

import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Collation;
import com.mongodb.client.model.CollationStrength;
import com.mongodb.client.model.CreateCollectionOptions;
import org.bson.Document;

public class CollationExample {
    public static void main(String[] args) {
        String nombreColeccion = "mi_coleccion";
        try (var mongoClient = MongoClients.create("mongodb://root:root@localhost:27017")) {

```



```

MongoDatabase database = mongoClient.getDatabase("mi_bd");
database.getCollection(nombreColeccion).drop();
// Configurar collation para la colección
Collation collation = Collation.builder()
    .locale("es") // Idioma español
    .caseLevel(false) // Ignorar mayúsculas
    .collationStrength(CollationStrength.PRIMARY) // Ignorar acentos
    .build();
// Crear la colección con el collation especificado
CreateCollectionOptions opciones = new CreateCollectionOptions()
    .capped(true) // Activar colección "capped" (tamaño limitado)
    .sizeInBytes(1024 * 1024) // Tamaño máximo en bytes (1 MB)
    .maxDocuments(10000) // Máximo de documentos permitidos
    .collation(collation); // Configuración del Collation
database.createCollection(nombreColeccion, opciones);
MongoCollection<Document> collection = database.getCollection(nombreColeccion);
// Insertar documentos en la colección
collection.insertOne(new Document("nombre", "café"));
collection.insertOne(new Document("nombre", "CAFÉ"));
collection.insertOne(new Document("nombre", "Cafe"));
collection.insertOne(new Document("nombre", "Cafetería"));
System.out.println("Datos insertados en la colección con collation.");
// Consultar datos con insensibilidad a acentos y mayúsculas
Document query = new Document("nombre", "cafe");
var cursor = collection.find(query).collation(collation);
System.out.println("Resultados de la consulta con collation: ");
cursor.forEach(document -> System.out.println( " - " + document.toJson()));
} catch (Exception e) {
    System.err.println("Error al interactuar con MongoDB: " + e.getMessage());
}
}
}

```

4.10 Eliminación de documentos.

El método `drop()` no elimina documentos individuales, sino que elimina toda la colección en *MongoDB*. Para eliminar documentos se pueden usar varios **métodos**: `deleteOne()`, `deleteMany()` y `findOneAndDelete()`. Tabla con **ejemplos** de eliminación de documentos de una colección:

Método	Descripción
<code>deleteOne(Filters.eq("campo", valor))</code>	Elimina el primer documento que cumple con el criterio específico.
<code>deleteMany(Filters.eq("campo", valor))</code>	Elimina todos los documentos que cumplen con el criterio específico.
<code>deleteMany(Filters.in("campo", valores))</code>	Elimina todos los documentos cuyos valores en el campo coinciden con alguno de los valores proporcionados.
<code>deleteMany(new Document())</code>	Elimina todos los documentos de la colección. Cuidado con este enfoque, ya que elimina todos los documentos.
<code>deleteMany(Filters.regex("campo", "expresión-regular"))</code>	Elimina todos los documentos donde el valor del campo coincide con una expresión regular.

`findOneAndDelete()` permite realizar de forma atómica la búsqueda de los documentos a eliminar y su posterior eliminación. A este método se le pasará el documento con los criterios, y de forma optativa las opciones de eliminación.

Los métodos de eliminación en *MongoDB* devuelven información diferente según el caso: `deleteOne()` y `deleteMany()` devuelven un objeto `DeleteResult`, del cual se puede obtener el número de documentos eliminados usando `getDeletedCount()`. Por otro lado, `findOneAndDelete()` devuelve el documento eliminado como un objeto `Document`, o `null` si no encuentra coincidencias; se puede acceder a su contenido con métodos como `toJson()`, `getString()`, `getInteger()`, `getDouble()`, entre otros.

Ejemplo: método que encuentra y elimina la primera persona que tiene el e-mail indicado.

```

public void eliminarPersona(MongoDatabase db, String email) {
    MongoCollection<Document> coleccion = db.getCollection("personas"); // Seleccionar la colección
    Document filtro = new Document("email", email); // Filtro con criterios de búsqueda
    try {
        Document eliminado = coleccion.findOneAndDelete(filtro);
        if (eliminado != null)

```

```

        System.out.println("Persona eliminada: " + eliminado.toJson());
    } else {
        System.out.println("No se encontró ninguna persona con el email proporcionado.");
    } catch (Exception e) {
        System.err.println("Error al eliminar la persona: " + e.getMessage());
    }
}

```

Ejemplo: programa que pide al usuario el nombre de la persona cuyo documento se debe eliminar de una colección.

```

import com.mongodb.client.*;
import com.mongodb.client.model.Filters;
import org.bson.Document;
import java.util.List;
import java.util.Scanner;

public class EliminarPersonaPorNombre {
    public static void main(String[] args) {
        String connectionString = "mongodb://root:root@localhost";

        try (Scanner scanner = new Scanner(System.in);
             MongoClient mongoClient = MongoClient.create(connectionString) {
            MongoDBDatabase database = mongoClient.getDatabase("mi_bd");
            MongoCollection<Document> personasColeccion = database.getCollection("personas2");

            // Insertar documentos de prueba (si la colección está vacía)
            if (personasColeccion.countDocuments() == 0) {
                insertarDocumentosDePrueba(personasColeccion);
                System.out.println("Se han insertado documentos de prueba.");
            }

            System.out.println("Personas en la colección:");
            mostrarDocumentos(personasColeccion); // Mostrar los documentos en la colección

            System.out.print("Introduce el nombre de la persona a eliminar: ");
            String nombre = scanner.nextLine().trim(); // Solicitar nombre de la persona a eliminar

            // Eliminar el primer documento que tenga el nombre proporcionado
            long eliminados = personasColeccion.deleteOne(Filters.eq("nombre", nombre)).getDeletedCount();
            if (eliminados > 0) {
                System.out.println("Persona eliminada correctamente.");
            } else {
                System.out.println("No se encontró ninguna persona con el nombre proporcionado.");
            }
        } catch (Exception e) {
            System.err.println("Error al interactuar con la BD: " + e.getMessage());
        }
    }

    // Método para insertar documentos de prueba en la colección
    private static void insertarDocumentosDePrueba(MongoCollection<Document> coleccion) {
        List<Document> documentos = List.of(
            new Document("nombre", "Juan").append("edad", 30).append("ciudad", "Granada"),
            new Document("nombre", "María").append("edad", 25).append("ciudad", "Sevilla"),
            new Document("nombre", "Carlos").append("edad", 35).append("ciudad", "Madrid"),
            new Document("nombre", "Ana").append("edad", 28).append("ciudad", "Málaga")
        );
        coleccion.insertMany(documentos);
    }

    // Método para mostrar los documentos en la colección
    private static void mostrarDocumentos(MongoCollection<Document> coleccion) {
        try (MongoCursor<Document> cursor = coleccion.find().iterator()) {
            if (!cursor.hasNext()) {
                System.out.println("No hay personas en la colección.");
                return;
            }
            while (cursor.hasNext()) {
                Document doc = cursor.next();
                System.out.println(doc.toJson());
            }
        }
    }
}

```

Cuando se realiza una **eliminación** en *MongoDB*, el resultado devuelve un objeto **DeleteResult**, el cual proporciona métodos para obtener información sobre la operación:

- **getDeletedCount()** → Devuelve el número de documentos eliminados.
- **wasAcknowledged()** → Indica si la operación fue reconocida por el servidor.

4.11 Actualización de documentos.

Para actualizar documentos se pueden usar varios métodos: **updateOne()**, **updateMany()** y **findOneAndUpdate()**,

que como se puede comprobar guardan un gran parecido con sus homólogos para eliminar documentos. Tabla con **ejemplos** de actualización de documentos de una colección:

Método	Descripción
<code>updateOne(Filters.eq("campo", valor), Updates.set("campo", nuevoValor))</code>	Actualiza el primer documento que cumple con el criterio especificado.
<code>updateOne(Filters.eq("campo", valor), Updates.combine(Updates.set("campo1", valor1), Updates.set("campo2", valor2)))</code>	Actualiza el primer documento que cumple con el criterio especificado con múltiples operaciones de actualización.
<code>updateMany(Filters.eq("campo", valor), Updates.set("campo", nuevoValor))</code>	Actualiza todos los documentos que cumplen con el criterio especificado.
<code>updateOne(Filters.eq("campo", valor), Updates.inc("campo", cantidad))</code>	Incrementa el valor de un campo numérico en una cantidad específica.
<code>updateOne(Filters.eq("campo", valor), Updates.push("campo", elemento))</code>	Agrega un elemento a un arreglo.
<code>updateOne(Filters.and(Filters.eq("campo", valor), Filters.gt("campoNum", 30)), Updates.set("campo", nuevoValor))</code>	Actualiza un documento solo si cumple con múltiples condiciones.
<code>updateOne(Filters.eq("campo", valor), Updates.set("campo", nuevoValor), new UpdateOptions().upsert(true))</code>	Actualiza el documento si existe, de lo contrario, inserta un nuevo documento.

findOneAndUpdate() permite realizar de forma atómica la búsqueda del documentos a actualizar y su posterior actualización. A este método se le pasará el documento con los criterios, la actualización y de forma optativa las opciones de actualización.

Ejemplo: método que encuentra y actualiza el nombre de la primera persona que tiene el e-mail indicado.

```
public void actualizarPersona(MongoDatabase db, String email, String nuevoNombre) {
    MongoCollection<Document> coleccion = db.getCollection("personas");
    Document documentoEncontrar = new Document("email", email);
    Document documentoActualizar = new Document("$set", new Document("nombre", nuevoNombre));

    try {
        Document documentoActualizado = coleccion.findOneAndUpdate(documentoEncontrar, documentoActualizar);
        if (documentoActualizado != null)
            System.out.println("Persona actualizada correctamente: " + documentoActualizado.toJson());
        else
            System.out.println("No se encontró una persona con el email proporcionado.");
    } catch (Exception e) {
        System.err.println("Error al actualizar la persona: " + e.getMessage());
    }
}
```

Ejemplo: reemplazar un documento completo.

```
db.getCollection("libros").replaceOne(new Document("_id", libro.getId()),
    new Document()
        .append("titulo", libro.getTitulo())
        .append("descripcion", libro.getDescripcion())
        .append("autor", libro.getAutor())
        .append("fecha", libro.getFecha())
        .append("disponible", libro.getDisponible()));
```

El método **replaceOne()** en *MongoDB* se usa para reemplazar un documento que coincida con un filtro específico. Devuelve un objeto de tipo **UpdateResult**.

Ejemplo: actualizar los documentos de acuerdo con un nombre y una nueva ciudad.

```
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoCursor;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Filters;
import com.mongodb.client.model.Updates;
import org.bson.Document;
import java.util.List;
import java.util.Scanner;

public class ActualizarPersona {
    public static void main(String[] args) {
        String connectionString = "mongodb://root:root@localhost";

        try (Scanner scanner = new Scanner(System.in);
            MongoClient mongoClient = MongoClients.create(connectionString) {
            MongoDatabase database = mongoClient.getDatabase("mi_bd");
            MongoCollection<Document> personasColeccion = database.getCollection("personas3");

            // Insertar documentos de prueba si la colección está vacía
            if (personasColeccion.countDocuments() == 0) {
                insertarDocumentosDePrueba(personasColeccion);
                System.out.println("Se han insertado documentos de prueba.");
            }
        }
```

Cuando se realiza una actualización en *MongoDB*, el resultado devuelve un objeto **UpdateResult**, el cual proporciona métodos para obtener información sobre la operación:

- **getMatchedCount()** → Devuelve el número de documentos que coincidieron con el filtro.
- **getModifiedCount()** → Devuelve el número de documentos que fueron realmente modificados.
- **getUpsertedId()** → Devuelve el ID del documento insertado si la operación fue un upsert (actualización o inserción).

```

System.out.println("Personas en la colección:");
mostrarDocumentos(personasColeccion); // Mostrar los documentos en la colección

System.out.print("\nIntroduce el nombre de la persona a actualizar: ");
String nombre = scanner.nextLine().trim(); // Pedir nombre de la persona a actualizar
if (nombre.isEmpty()) {
    System.out.println("El nombre no puede estar vacío.");
    return;
}

System.out.print("Introduce la nueva ciudad para esta persona: ");
String nuevaCiudad = scanner.nextLine().trim(); // Pedir nuevo valor para la ciudad
if (nuevaCiudad.isEmpty()) {
    System.out.println("La ciudad no puede estar vacía.");
    return;
}

// 1. Usar updateOne() para actualizar solo el primer documento encontrado
System.out.println("\nUsando updateOne():");
long actualizadosUno = personasColeccion.updateOne(
    Filters.eq("nombre", nombre),
    Updates.set("ciudad", nuevaCiudad)
).getModifiedCount();
if (actualizadosUno > 0)
    System.out.println("Se actualizó correctamente el primer documento encontrado.");
else
    System.out.println("No se encontró ninguna persona con el nombre proporcionado.");

// 2. Usar updateMany() para actualizar todos los documentos que coincidan
System.out.println("\nUsando updateMany():");
long actualizadosMuchos = personasColeccion.updateMany(
    Filters.eq("nombre", nombre),
    Updates.set("ciudad", nuevaCiudad)
).getModifiedCount();
System.out.println("Se actualizaron " + actualizadosMuchos + " documento(s).");

// 3. Usar findOneAndUpdate() para encontrar y actualizar un documento, devolviendo el original
System.out.println("\nUsando findOneAndUpdate():");
Document documentoActualizado = personasColeccion.findOneAndUpdate(
    Filters.eq("nombre", nombre),
    Updates.set("ciudad", nuevaCiudad)
);
if (documentoActualizado != null) {
    System.out.println("Documento original actualizado:");
    System.out.println(documentoActualizado.toJson());
} else
    System.out.println("No se encontró ninguna persona con el nombre proporcionado.");

System.out.println("\nPersonas actualizadas en la colección:");
mostrarDocumentos(personasColeccion); // Mostrar los documentos actualizados
} catch (Exception e) {
    System.err.println("Error al interactuar con la BD: " + e.getMessage());
}
}

private static void insertarDocumentosDePrueba(MongoCollection<Document> coleccion) {
    List<Document> documentos = List.of(
        new Document("nombre", "Juan").append("edad", 30).append("ciudad", "Granada"),
        new Document("nombre", "María").append("edad", 25).append("ciudad", "Sevilla"),
        new Document("nombre", "Carlos").append("edad", 35).append("ciudad", "Madrid"),
        new Document("nombre", "Ana").append("edad", 28).append("ciudad", "Málaga"),
        new Document("nombre", "Juan").append("edad", 40).append("ciudad", "Almería") // duplicado
    );
    coleccion.insertMany(documentos);
}

private static void mostrarDocumentos(MongoCollection<Document> coleccion) {
    try (MongoCursor<Document> cursor = coleccion.find().iterator()) {
        if (!cursor.hasNext()) {
            System.out.println("No hay personas en la colección.");
            return;
        }
        while (cursor.hasNext()) {
            Document doc = cursor.next();
            System.out.println(doc.toJson());
        }
    }
}

```

```

    }
}

```

Ejemplo: *CRUD* para una colección de productos.

➤ **Producto.java**

```

import java.time.LocalDate;

public class Producto {
    private String id;
    private String nombre;
    private double precio;
    private int stock;
    private LocalDate fechaAlta;

    // Constructor
    public Producto(String id, String nombre, double precio, int stock, LocalDate fechaAlta) {
        this.id = id;
        this.nombre = nombre;
        this.precio = precio;
        this.stock = stock;
        this.fechaAlta = fechaAlta;
    }

    // Getters
    public String getId() { return id; }
    public String getNombre() { return nombre; }
    public double getPrecio() { return precio; }
    public int getStock() { return stock; }
    public LocalDate getFechaAlta() { return fechaAlta; }

    // Setters
    public void setNombre(String nombre) { this.nombre = nombre; }
    public void setPrecio(double precio) { this.precio = precio; }
    public void setStock(int stock) { this.stock = stock; }
    public void setFechaAlta(LocalDate fechaAlta) { this.fechaAlta = fechaAlta; }
}

```

➤ **ProductoService.java**

```

import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.result.UpdateResult;
import org.bson.Document;
import com.mongodb.client.model.Filters;

public class ProductoService {
    private MongoCollection<Document> coleccion;

    public ProductoService(MongoDatabase db) { this.coleccion = db.getCollection("productos"); }

    public void crearProducto(Producto producto) { // Crear un producto
        Document doc = new Document("_id", producto.getId())
            .append("nombre", producto.getNombre())
            .append("precio", producto.getPrecio())
            .append("stock", producto.getStock())
            .append("fechaAlta", producto.getFechaAlta().toString()); // Convertir LocalDate a String
        coleccion.insertOne(doc);
        System.out.println("Producto creado correctamente.");
    }

    public void mostrarProducto(String id) { // Leer un producto por su ID
        Document producto = coleccion.find(Filters.eq("_id", id)).first();
        if (producto != null)
            System.out.println("Producto encontrado: " + producto.toJson());
        else
            System.out.println("No se encontró el producto con el ID: " + id);
    }

    public void actualizarProducto(String id, Producto nuevoProducto) { // Actualizar un producto
        Document updatedProduct = new Document()
            .append("nombre", nuevoProducto.getNombre())
            .append("precio", nuevoProducto.getPrecio())
            .append("stock", nuevoProducto.getStock())
            .append("fechaAlta", nuevoProducto.getFechaAlta().toString());
        UpdateResult resultado = coleccion.updateOne(Filters.eq("_id", id), new Document("$set", updatedProduct));
        if (resultado.getMatchedCount() > 0)
            System.out.println("Producto actualizado correctamente.");
        else

```



```

        System.out.println("No se encontró el producto con el ID: " + id);
    }
    public void eliminarProducto(String id) { // Eliminar un producto
        long eliminados = coleccion.deleteOne(Filters.eq("_id", id)).getDeletedCount();
        if (eliminados > 0)
            System.out.println("Producto eliminado correctamente.");
        else
            System.out.println("No se encontró el producto con el ID: " + id);
    }
    public void mostrarTodosLosProductos() { // Mostrar todos los productos
        coleccion.find().forEach(doc -> System.out.println(doc.toJson()));
    }
}

```

➤ Main.java

```

import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoDatabase;
import java.time.LocalDate;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        String cs = "mongodb://root:root@localhost:27017";
        try (MongoClient mongoClient = MongoClients.create(cs)) {
            MongoDatabase db = mongoClient.getDatabase("tienda");
            ProductoService servicioProducto = new ProductoService(db);
            Scanner scanner = new Scanner(System.in);
            boolean salir = false;
            while (!salir) {
                System.out.println("\n┌─── Menú CRUD de productos ──┐");
                System.out.println("1. Crear producto.                ");
                System.out.println("2. Mostrar producto.             ");
                System.out.println("3. Actualizar producto.          ");
                System.out.println("4. Eliminar producto.            ");
                System.out.println("5. Mostrar todos los productos.  ");
                System.out.println("0. Salir.                        ");
                System.out.println("└────────────────────────────────┘");
                System.out.print("Selecciona una opción: ");
                int opcion = scanner.nextInt();
                scanner.nextLine(); // Limpiar buffer
                switch (opcion) {
                    case 1: // Crear producto
                        System.out.print("Introduce el ID del producto: ");
                        String idCrear = scanner.nextLine();
                        System.out.print("Introduce el nombre del producto: ");
                        String nombreCrear = scanner.nextLine();
                        System.out.print("Introduce el precio del producto: ");
                        double precioCrear = scanner.nextDouble();
                        System.out.print("Introduce el stock del producto: ");
                        int stockCrear = scanner.nextInt();
                        LocalDate fechaAlta = LocalDate.now();
                        Producto productoCrear = new Producto(idCrear, nombreCrear, precioCrear,
                                                            stockCrear, fechaAlta);
                        servicioProducto.crearProducto(productoCrear);
                        break;
                    case 2: // Mostrar producto
                        System.out.print("Introduce el ID del producto a mostrar: ");
                        String idLeer = scanner.nextLine();
                        servicioProducto.mostrarProducto(idLeer);
                        break;
                    case 3: // Actualizar producto
                        System.out.print("Introduce el ID del producto a actualizar: ");
                        String idActualizar = scanner.nextLine();
                        System.out.print("Introduce el nuevo nombre del producto: ");
                        String nuevoNombre = scanner.nextLine();
                        System.out.print("Introduce el nuevo precio del producto: ");
                        double nuevoPrecio = scanner.nextDouble();
                        System.out.print("Introduce el nuevo stock del producto: ");
                        int nuevoStock = scanner.nextInt();
                        Producto productoActualizar = new Producto(idActualizar, nuevoNombre,
                                                                nuevoPrecio, nuevoStock, LocalDate.now());

```



```

        servicioProducto.actualizarProducto(idActualizar, productoActualizar);
        break;
    case 4: // Eliminar producto
        System.out.print("Introduce el ID del producto a eliminar: ");
        String idEliminar = scanner.nextLine();
        servicioProducto.eliminarProducto(idEliminar);
        break;
    case 5: // Mostrar todos los productos
        servicioProducto.mostrarTodosLosProductos();
        break;
    case 0: // Salir
        salir = true;
        System.out.println("Saliendo del programa.");
        break;
    default:
        System.out.println("Opción no válida.");
    }
}
}
}
}
}

```

4.12 Framework de agregación.

En *MongoDB*, el *framework* de agregación **permite realizar operaciones de procesamiento y transformación de datos** en colecciones (filtrado, transformación, agrupación, etc.).

Las **operaciones relacionadas con el *framework* de agregación** se realizan con el método `aggregate()`. A este método se le **proporcionará el pipeline de operaciones** (`$match`, `$project`, `$group`, `$sort`, `$lookup`, ...) en forma de **lista**. El método **devolverá un objeto `AggregationIterable`** que se podrá recorrer como un cursor.

El operador `$lookup` se utiliza para realizar operaciones de "LEFT JOIN" entre dos colecciones (en *BD* relacionales retorna todos los registros de la "tabla izquierda" y los registros coincidentes de la "tabla derecha", si no hay coincidencias en la "tabla derecha" para un registro en la "tabla izquierda", se devolverán valores nulos para las columnas de la "tabla derecha"). Permite combinar documentos de la colección de origen con documentos de otra colección basándose en campos específicos. Esta operación es especialmente **útil cuando se necesita obtener información de documentos relacionados en diferentes colecciones**.

Tabla con algunos de los **métodos** más usados de la **clase `AggregationIterable`**:

Método	Descripción
<code>void forEach(Block<? super T> block)</code>	Itera sobre los documentos de la agregación y ejecuta el bloque proporcionado en cada documento.
<code>MongoCursor<T> iterator()</code>	Devuelve un iterador para recorrer los documentos de la agregación.
<code>T first()</code>	Devuelve el primer documento del resultado de la agregación, o null si no hay documentos.
<code>void toCollection()</code>	Escribe los resultados de la agregación en una colección.
<code>AggregationIterable<T> allowDiskUse(Boolean allow)</code>	Permite que MongoDB use el almacenamiento en disco para operaciones que exceden la memoria RAM.
<code>AggregationIterable<T> maxTime(long maxTime, TimeUnit timeUnit)</code>	Especifica el tiempo máximo permitido para que se ejecute la operación de agregación.
<code>AggregationIterable<T> hint(Document hint)</code>	Proporciona un índice sugerido para usar en la agregación.
<code>AggregationIterable<T> comment(String comment)</code>	Agrega un comentario a la operación de agregación para propósitos de depuración o análisis.

Ejemplo: método que devuelve el número total de actores por género de *USA*.

```

public void totalDeActoresPorGenero(MongoDatabase db) {
    MongoCollection<Document> coleccion = db.getCollection("actores"); // Seleccionar la colección actores
    // Crear el pipeline de operaciones, primero con $match para filtrar por país
    Document match = new Document("$match", new Document("pais", "USA"));
    // Operación $group para agrupar por género y contar el total
    Document camposAgrupacion = new Document( "_id", "$genero").append("total", new Document( "$sum", 1));
    Document grupo = new Document("$group", camposAgrupacion);
    List<Document> pipeline = Arrays.asList(match, grupo); // Construir el pipeline
    MongoCursor<Document> cursor = coleccion.aggregate(pipeline).iterator(); // Ejecutar la agregación
    System.out.println("*** Total de actores por género de USA ***");
}

```

```

while (cursor.hasNext()) {
    Document documento = cursor.next();
    System.out.println(" - " + documento.getString("_id") + ": " + documento.getInteger("total"));
}
}

```

Ejemplo: mostrar la edad media por ciudad de aquellos amigos que tengan más de 25 años.

```

import com.mongodb.client.AggregateIterable;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import org.bson.Document;
import java.util.Arrays;

public class AggregationFrameworkEjemplo {
    public static void main(String[] args) {
        String cs = "mongodb://root:root@localhost";
        try (var mongoClient = MongoClients.create(cs)) {
            MongoDatabase database = mongoClient.getDatabase("mi_bd");
            MongoCollection<Document> coleccion = database.getCollection("amigos");

            // Insertar documentos de ejemplo en la colección
            coleccion.insertMany(Arrays.asList(
                new Document("nombre", "Juan").append("edad", 15).append("ciudad", "Granada"),
                new Document("nombre", "Alicia").append("edad", 28).append("ciudad", "Málaga"),
                new Document("nombre", "Carlos").append("edad", 35).append("ciudad", "Granada"),
                new Document("nombre", "Luis").append("edad", 35).append("ciudad", "Granada"),
                new Document("nombre", "Roberto").append("edad", 13).append("ciudad", "Almería"),
                new Document("nombre", "Isabel").append("edad", 35).append("ciudad", "Granada"),
                new Document("nombre", "Eva").append("edad", 27).append("ciudad", "Málaga")));

            // Pipeline de agregación:
            AggregateIterable<Document> result = coleccion.aggregate(Arrays.asList(
                // Filtrar documentos donde 'edad' sea mayor que 25
                new Document("$match", new Document("edad", new Document("$gt", 25))),
                // Agrupar por 'ciudad' y calcular el promedio de 'edad'
                new Document("$group", new Document("_id", "$ciudad").append("mediaEdad",
                    new Document("$avg", "$edad"))),

                // Ordenar por 'mediaEdad' de forma descendente
                new Document("$sort", new Document("mediaEdad", -1)),
                // Proyectar solo '_id' y 'mediaEdad'
                new Document("$project", new Document("_id", 1).append("mediaEdad", 1))));

            // Mostrar resultados por pantalla
            result.forEach(document -> System.out.println(document.toJson()));
        }
    }
}

```

Ejemplo: operación de agregación que combina datos de dos colecciones mediante el operador \$lookup.

```

import com.mongodb.client.AggregateIterable;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import org.bson.Document;
import java.util.Arrays;

public class AgregacionMultiplesColeccionesEjemplo {
    public static void main(String[] args) {
        String cs = "mongodb://root:root@localhost";
        try (var mongoClient = MongoClients.create(cs)) {
            MongoDatabase database = mongoClient.getDatabase("mi_bd");

            // Insertar algunos documentos de ejemplo en la colección clientes
            MongoCollection<Document> clientesCollection = database.getCollection("clientes");
            clientesCollection.insertMany(Arrays.asList(
                new Document("nombre", "Juan").append("edad", 30),
                new Document("nombre", "Alicia").append("edad", 28),
                new Document("nombre", "Francisco").append("edad", 35),
                new Document("nombre", "Eva").append("edad", 27)));

            // Insertar algunos documentos de ejemplo en la colección pedidos
            MongoCollection<Document> pedidosCollection = database.getCollection("pedidos");
            pedidosCollection.insertMany(Arrays.asList(
                new Document("numeroPedido", "123").append("nombrePersona", "Juan").append("cantidad", 150),
                new Document("numeroPedido", "124").append("nombrePersona", "Alicia").append("cantidad", 200),
                new Document("numeroPedido", "125").append("nombrePersona", "Juan").append("cantidad", 100),

```

```

        new Document("numeroPedido", "126").append("nombrePersona", "Francisco").append("cantidad", 300)));
// Pipeline de agregación que combina datos de ambas colecciones
AggregateIterable<Document> result = clientesCollection.aggregate(Arrays.asList(
    new Document("$lookup", new Document() // Realizar 'lookup' para combinar clientes y pedidos
        .append("from", "pedidos")
        .append("localField", "nombre")
        .append("foreignField", "nombrePersona")
        .append("as", "pedidos")),
    // Proyectar solo los campos deseados
    new Document("$project", new Document("_id", 0)
        .append("nombre", 1)
        .append("edad", 1)
        .append("pedidos",
            new Document("$map", new Document()
                .append("input", "$pedidos")
                .append("as", "pedido")
                .append("in", new Document("numeroPedido", "$$pedido.numeroPedido")
                    .append("cantidad", "$$pedido.cantidad")
                )
            )
        )
    ),
    // Agrupar por nombre para eliminar duplicados
    new Document("$group", new Document("_id", "$nombre")
        .append("nombre", new Document("$first", "$nombre"))
        .append("edad", new Document("$first", "$edad"))
        .append("pedidos", new Document("$first", "$pedidos"))
    )
));
// Mostrar resultados por pantalla
result.forEach(document -> System.out.println(document.toJson()));
}
}
}

```

Más información en <https://www.mongodb.com/docs/drivers/java/sync/current/quick-start/>

4.13 Acceder a una colección con un *POJO*.

Un **POJO** es una **clase ordinaria de Java** que no está sujeta a restricciones de *framework* o tecnología específica.

Para **acceder a una colección** se usarán las **clases** `MongoDatabase` y `MongoCollection`. La primera de ellas para acceder a la *BD* y la segunda para acceder a la colección.

Adicionalmente, se usarán también las **clases** `CodecProvider` y `CodecRegistry` que están **relacionadas con la serialización y deserialización de documentos BSON en objetos Java**.

MongoDB utiliza `CodecProvider` para **obtener códecs para las clases utilizadas en la aplicación**. Cuando se inserta un objeto *POJO* en una colección de *MongoDB*, se necesita un códec para convertir el objeto *POJO* a un documento *BSON* que se puede almacenar en la *BD*.

`CodecRegistry` actúa como un **registro centralizado de códecs** que se puede utilizar para realizar conversiones entre objetos *Java* y documentos *BSON* en *MongoDB*.

Hay dos **formas de manejar la serialización y deserialización en MongoDB**:

1. **Usando códec personalizado** (sin anotaciones): ofrece un control total sobre cómo los objetos se serializan y deserializan (ver ejemplo).
 - Implementar una clase códec que defina cómo convertir los objetos a *BSON* y viceversa.
 - Crear un `CodecProvider` para registrar el códec para tipos específicos.

Es útil cuando se necesita una lógica personalizada para la serialización/deserialización o si se trabaja con tipos complejos que no se manejan automáticamente.
2. **Usando anotaciones** (más sencillo): es más simple y directo, sin necesidad de escribir código adicional para serializar los objetos.
 - Utilizar anotaciones específicas como `@BsonProperty`, `@BsonId`, etc., en las clases que se desean

serializar.

Anotación	Descripción
@BsonId	Indica que el campo es el identificador único (<code>_id</code>) en MongoDB.
@BsonProperty	Permite especificar el nombre del campo BSON en MongoDB diferente al nombre del campo en la clase Java.
@BsonIgnore	Indica que el campo no debe incluirse al mapear el objeto a BSON.
@BsonCreator	Indica un constructor o método de fábrica que debe utilizarse para crear instancias de la clase a partir de BSON.
@BsonDiscriminator	Define un campo discriminador que se utiliza para determinar la clase concreta cuando se realiza el mapeo de la herencia (polimorfismo).
@BsonProperty	Anota un método getter para especificar el nombre del campo BSON en MongoDB diferente al nombre del método.
@BsonRepresentation	Especifica la representación BSON a utilizar para el campo (Ej.: <code>BsonType.DOCUMENT</code> , <code>BsonType.STRING</code>).
@BsonSerialized	Anota un método getter para proporcionar una expresión para la serialización personalizada.

- *MongoDB* se encarga automáticamente de mapear los objetos a *BSON* y viceversa utilizando las anotaciones.

Es ideal si no se necesita una lógica personalizada de serialización/deserialización y si los objetos siguen una estructura simple.

Las **anotaciones son necesarias únicamente en algunos casos específicos** donde se desea aprovechar ciertas capacidades de *frameworks* o bibliotecas que requieren metadatos adicionales para entender cómo manejar las clases, como, por ejemplo:

- **Mapeo personalizado de campos** (field mapping): si los nombres de los campos en la clase *Java* son diferentes a los nombres en *MongoDB*, las anotaciones permiten mapearlos explícitamente, por ejemplo, con `@BsonProperty`.
- **Compatibilidad con POJO's** (Plain Old Java Objects): cuando se utiliza `PojoCodecProvider` para manejar automáticamente las clases, las anotaciones ayudan a indicar cómo decodificar/serializar los objetos. Por ejemplo, si una clase no tiene un constructor vacío o se tiene un constructor personalizado, se puede usar `@BsonCreator` para especificar cuál usar.
- **Control de serialización**: si se necesita omitir ciertos campos de la serialización (como contraseñas o datos sensibles), se puede usar anotaciones como `@BsonIgnore`.
- **Definir un campo como `_id`**: por defecto, *MongoDB* asigna un campo `_id` a cada documento. Si la clase tiene su propio campo que se desea usar como clave primaria, se puede indicar con `@BsonId`.
- **Convivir con otras estructuras**: si se tiene un diseño más complejo, como documentos anidados o listas de objetos, las anotaciones ayudan a definir cómo se deben manejar esas relaciones.

No se necesitan anotaciones si:

- Se usan los mismos nombres para las propiedades de las clases *Java* y los campos en *MongoDB*.
- No se tienen requisitos especiales para los campos (como usar un id personalizado o ignorar campos).
- Se está usando el `PojoCodecProvider` con `.automatic(true)`.

Por lo general, la segunda opción es más fácil de usar en la mayoría de los casos, mientras que los códec personalizados son útiles cuando se necesita un control detallado sobre cómo los objetos se transforman en *BSON*.

Código necesario para **trabajar con CodecProvider y CodecRegistry**:

```
import com.mongodb.ConnectionString;
import com.mongodb.MongoClientSettings;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoDatabase;
import org.bson.codecs.configuration.CodecRegistries;
import org.bson.codecs.configuration.CodecRegistry;
import org.bson.codecs.pojo.PojoCodecProvider;

public class MongoDBCodecRegistry {
    public static void main(String[] args) {
        CodecRegistry cr = CodecRegistries.fromRegistries(MongoClientSettings.getDefaultCodecRegistry(),
            CodecRegistries.fromProviders(PojoCodecProvider.builder().automatic(true).build()));
        // Crear un cliente MongoDB con el CodecRegistry personalizado
        MongoClientSettings settings = MongoClientSettings.builder()
            .applyConnectionString(new ConnectionString("mongodb://root:root@localhost"))
            .codecRegistry(cr)
            .build();
    }
}
```

```

MongoClient mongoClient = MongoClient.create(settings);
MongoDatabase database = mongoClient.getDatabase("mi_bd");

// Resto de la lógica de la aplicación aquí ...

mongoClient.close(); // Cerrar la conexión cliente MongoDB
}
}

```

Ejemplo: insertar documento persona.

➤ **Persona.java**

```

import org.bson.codecs.pojo.annotations.*;
import org.bson.types.ObjectId;

public class Persona {
    @BsonId
    private ObjectId id;
    @BsonProperty("nombre")
    private String nombre;
    @BsonProperty("edad")
    private int edad;

    @BsonCreator
    public Persona(@BsonProperty("nombre") String nombre, @BsonProperty("edad") int edad) {
        this.nombre = nombre; this.edad = edad;
    }

    public ObjectId getId() { return id; }
    public String getNombre() { return nombre; }
    public int getEdad() { return edad; }
    public void setId(ObjectId id) { this.id = id; }
    public void setNombre(String nombre) { this.nombre = nombre; }
    public void setEdad(int edad) { this.edad = edad; }

    @Override
    public String toString() {
        return "Persona{" + "id=" + id + ", nombre=" + nombre + '\'' + ", edad=" + edad + '\'';
    }
}

```

Para que el mapeo automático con `automatic(true)` funcione sin anotaciones, es necesario que los nombres de las propiedades de la clase coincidan con los campos en *MongoDB*, que exista un constructor vacío público para la deserialización, y que los tipos de datos (como `ObjectId` para el campo `_id`) sean correctos. Las anotaciones solo son necesarias si se desea cambiar nombres, manejar constructores específicos o ajustar comportamientos especiales. En *Persona.java* se podría prescindir del uso de las anotaciones si se agrega un constructor vacío.

Tener en cuenta que este ejemplo usa la anotación `@BsonId` para indicar el campo que se utilizará como identificador único (`_id`) en *MongoDB*. Se puede personalizar más el mapeo utilizando otras anotaciones de `org.bson.codecs.pojo.annotations`.

Un **record** en *Java* es una **clase inmutable** (una vez que se crea un objeto record, sus campos no pueden modificarse) **diseñada para almacenar datos, simplificando la creación de objetos al generar automáticamente el constructor, los métodos `getter`, `toString()`, `hashCode()` y `equals()`**. Se puede usar con *MongoDB* mediante anotaciones como `@BsonId` y `@BsonProperty`, permitiendo la compatibilidad con las operaciones de *BD* sin necesidad de modificar el código que gestiona la inserción y actualización de datos. Esto ofrece una forma más breve y eficiente de manejar datos inmutables en *Java*.

```

import org.bson.codecs.pojo.annotations.BsonId;
import org.bson.codecs.pojo.annotations.BsonProperty;
import org.bson.types.ObjectId;

public record Persona(
    @BsonId ObjectId id,
    @BsonProperty("nombre") String nombre,
    @BsonProperty("edad") int edad) {}

```

Los records no tienen métodos setter porque sus campos son inmutables una vez que el objeto es creado.

➤ **MongoDBCodecRegistry.java**

Suponiendo que se tiene un objeto persona de la clase *Persona* que se quiere insertar como documento en una colección *MongoDB*, se podría hacer lo siguiente:

```

import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;

...

// Obtener la BD y la colección
MongoDatabase database = mongoClient.getDatabase("mi_bd");
MongoCollection<Persona> personaColeccion = database.getCollection("personas", Persona.class);

// Insertar una persona en la colección
Persona personaAInsertar = new Persona("Juan Moreno", 30);
personaColeccion.insertOne(personaAInsertar);

```


Ejemplo: búsqueda utilizando la clase Persona.

- Persona.java
- MongoDBCodecRegistry.java

```
...
MongoCollection<Persona> coleccionPersonas = db.getCollection("personas", Persona.class);
List<Persona> personas = coleccionPersonas.find().into(new ArrayList<Persona>());
...
```

De esta forma se obtendrá, mapeando automáticamente a través del CodecRegistry, una colección de objetos Persona directamente de una colección *MongoDB*.

Ejemplo: configuración del CodecRegistry, inserción de objetos en la colección y búsqueda utilizando la clase Persona.

- Persona.java
- MongoDBCodecRegistry.java

```
import com.mongodb.ConnectionString;
import com.mongodb.MongoClientSettings;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Projections;
import com.mongodb.client.model.Sorts;
import org.bson.codecs.configuration.CodecRegistries;
import org.bson.codecs.configuration.CodecRegistry;
import org.bson.codecs.pojo.PojoCodecProvider;
import java.util.ArrayList;
import java.util.List;

public class MongoDBCodecRegistry {
    public static void main(String[] args) {
        CodecRegistry cr = CodecRegistries.fromRegistries(MongoClientSettings.getDefaultCodecRegistry(),
            CodecRegistries.fromProviders(PojoCodecProvider.builder().automatic(true).build()));
        // Crear un cliente MongoDB con el CodecRegistry personalizado
        MongoClientSettings settings = MongoClientSettings.builder()
            .applyConnectionString(new ConnectionString("mongodb://root:root@localhost"))
            .codecRegistry(cr)
            .build();

        MongoClient mongoClient = MongoClients.create(settings);
        MongoDatabase database = mongoClient.getDatabase("mibd");
        MongoCollection<Persona> personasColeccion = database.getCollection("personas", Persona.class);
        // Insertar solo si la colección está vacía (para evitar duplicados en este ejemplo)
        if (personasColeccion.countDocuments() == 0) {
            // Crear objetos Persona
            Persona persona1 = new Persona("Juan", 25);
            Persona persona2 = new Persona("Maria", 30);
            // Insertar en la colección
            personasColeccion.insertMany(List.of(persona1, persona2));
            System.out.println("Personas insertadas en la colección.");
        }

        // Búsqueda de todas las personas en la colección
        System.out.println("Todas las personas de la colección:");
        // Volcar los resultados en una lista usando into()
        List<Persona> personasList = new ArrayList<>();
        personasColeccion.find()
            .projection(Projections.excludeId()) // Excluir el campo _id de la salida
            .sort(Sorts.ascending("nombre")) // Ordenar por nombre ascendente
            .into(personasList);
        personasList.forEach(System.out::println); // Imprimir la lista de personas
        mongoClient.close(); // Cerrar la conexión cliente MongoDB
    }
}
```

Ejemplo: eliminar documento persona.

- Persona.java
- MongoDBCodecRegistry.java

```
...
MongoCollection<Persona> coleccionPersonas = db.getCollection("personas", Persona.class);
```



```
long eliminados = coleccionPersonas.deleteOne(Filters.eq("nombre", persona.getNombre())).getDeletedCount();
...
```

Ejemplo: modificar un documento persona.

- Persona.java
- MongoDBCodecRegistry.java

```
...
MongoCollection<Persona> coleccionPersonas = db.getCollection("personas", Persona.class);
coleccionPersonas.replaceOne(Filters.eq("nombre", persona.getNombre()), persona);
...
```

Ejemplo: uso de replaceOne para reemplazar un documento.

- Persona.java
- ReemplazarPersona.java

```
import com.mongodb.ConnectionString;
import com.mongodb.MongoClientSettings;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Filters;
import org.bson.codecs.configuration.CodecRegistries;
import org.bson.codecs.configuration.CodecRegistry;
import org.bson.codecs.pojo.PojoCodecProvider;
import java.util.Scanner;

public class ReemplazarPersona {
    public static void main(String[] args) {
        String connectionString = "mongodb://root:root@localhost";
        PojoCodecProvider pojoCodecProvider = PojoCodecProvider.builder().automatic(true).build();
        CodecRegistry cr = CodecRegistries.fromRegistries(MongoClientSettings.getDefaultCodecRegistry(),
            CodecRegistries.fromProviders(pojoCodecProvider));

        // Crear un cliente MongoDB con el CodecRegistry personalizado
        MongoClientSettings settings = MongoClientSettings.builder()
            .applyConnectionString(new ConnectionString(connectionString))
            .codecRegistry(cr)
            .build();

        // Solicitar al usuario los detalles de la persona a reemplazar
        Scanner scanner = new Scanner(System.in);
        System.out.print("Introduce el nombre de la persona a reemplazar: ");
        String nombreReemplazar = scanner.nextLine();
        System.out.print("Introduce el nuevo nombre de la persona a reemplazar: ");
        String nuevoNombreReemplazar = scanner.nextLine();
        System.out.print("Introduce la nueva edad: ");
        int nuevaEdad = scanner.nextInt();

        try (var mongoClient = MongoClients.create(settings)) {
            MongoDatabase database = mongoClient.getDatabase("mibd");
            MongoCollection<Persona> personaCollection = database.getCollection("personas2", Persona.class);

            // Crear un nuevo objeto Persona con los detalles proporcionados
            Persona nuevaPersona = new Persona(nuevoNombreReemplazar, nuevaEdad);

            // Reemplazar el documento que coincide con el nombre proporcionado
            personaCollection.replaceOne(Filters.eq("nombre", nombreReemplazar), nuevaPersona);
            System.out.println("Persona reemplazada correctamente.");
        }
    }
}
```

Ejemplo: no usar anotaciones en la clase Persona y trabajar con una clase Codec personalizada.

- Persona.java

```
import org.bson.types.ObjectId;

public class Persona {
    private ObjectId id;
    private String nombre;
    private int edad;

    public Persona() {}
    public Persona(String nombre, int edad) { this.nombre = nombre; this.edad = edad; }
    public ObjectId getId() { return id; }
    public String getNombre() { return nombre; }
```

```

public int getEdad() { return edad; }
public void setId(ObjectId id) { this.id = id; }
public void setNombre(String nombre) { this.nombre = nombre; }
public void setEdad(int edad) { this.edad = edad; }

@Override
public String toString() {
    return "Persona{" + "id=" + id + ", nombre='" + nombre + '\'' + ", edad=" + edad + '}';
}
}

```

➤ PersonaCodec.java

```

import org.bson.*;
import org.bson.codecs.Codec;
import org.bson.codecs.DecoderContext;
import org.bson.codecs.EncoderContext;
import org.bson.types.ObjectId;

public class PersonaCodec implements Codec<Persona> {
    @Override
    public Persona decode(BsonReader reader, DecoderContext decoderContext) {
        ObjectId id = null;
        String nombre = null;
        int edad = 0;

        reader.readStartDocument();
        while (reader.readBsonType() != BsonType.END_OF_DOCUMENT) {
            String fieldName = reader.readName();
            switch (fieldName) {
                case "_id" -> id = reader.readObjectId();
                case "nombre" -> nombre = reader.readString();
                case "edad" -> edad = reader.readInt32();
                default -> reader.skipValue(); // Ignorar campos desconocidos
            }
        }
        reader.readEndDocument();

        Persona persona = new Persona(nombre, edad);
        persona.setId(id); // Configurar el ID manualmente
        return persona;
    }

    @Override
    public void encode(BsonWriter writer, Persona value, EncoderContext encoderContext) {
        writer.writeStartDocument();
        if (value.getId() != null)
            writer.writeObjectId("_id", value.getId());
        writer.writeString("nombre", value.getNombre());
        writer.writeInt32("edad", value.getEdad());
        writer.writeEndDocument();
    }

    @Override
    public Class<Persona> getEncoderClass() {
        return Persona.class;
    }
}

```

➤ Main.java

```

import com.mongodb.MongoClientSettings;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import org.bson.codecs.configuration.CodecRegistry;
import org.bson.codecs.configuration.CodecRegistries;

public class Main {
    public static void main(String[] args) {
        CodecRegistry codecRegistry = CodecRegistries.fromRegistries(
            MongoClientSettings.getDefaultCodecRegistry(),
            CodecRegistries.fromCodecs(new PersonaCodec())
        );

        MongoClientSettings settings = MongoClientSettings.builder()
            .applyConnectionString(new com.mongodb.ConnectionString("mongodb://root:root@localhost:27017"))
            .codecRegistry(codecRegistry)
    }
}

```

```

        .build();
        MongoClient mongoClient = MongoClient.create(settings);
        MongoDB database = mongoClient.getDatabase("mi_bd");
        MongoCollection<Persona> collection = database.getCollection("personas2", Persona.class);
        Persona persona = new Persona("Juan", 30);
        collection.insertOne(persona);
        Persona personaLeida = collection.find().first();
        System.out.println("Persona leída: " + personaLeida);
    }
}

```

Ejemplo: CRUD de productos.

➤ Producto.java

```

import org.bson.codecs.pojo.annotations.BsonId;
import org.bson.codecs.pojo.annotations.BsonProperty;
import org.bson.types.ObjectId;
import java.time.LocalDate;

public class Producto {
    @BsonId
    private ObjectId id; // Este es el ObjectId de MongoDB
    @BsonProperty("idProducto")
    private int idProducto; // ID entero para usar en las operaciones CRUD
    @BsonProperty("nombre")
    private String nombre;
    @BsonProperty("precio")
    private double precio;
    @BsonProperty("stock")
    private int stock;
    @BsonProperty("fechaAlta")
    private LocalDate fechaAlta;

    public Producto() { } // Constructor vacío (sin parámetros) requerido

    // Constructor con parámetros
    public Producto(int idProducto, String nombre, double precio, int stock, LocalDate fechaAlta) {
        this.idProducto = idProducto;
        this.nombre = nombre;
        this.precio = precio;
        this.stock = stock;
        this.fechaAlta = fechaAlta;
    }

    // Getters y setters
    public ObjectId getId() { return id; }
    public void setId(ObjectId id) { this.id = id; }
    public int getIdProducto() { return idProducto; }
    public void setIdProducto(int idProducto) { this.idProducto = idProducto; }
    public String getNombre() { return nombre; }
    public void setNombre(String nombre) { this.nombre = nombre; }
    public double getPrecio() { return precio; }
    public void setPrecio(double precio) { this.precio = precio; }
    public int getStock() { return stock; }
    public void setStock(int stock) { this.stock = stock; }
    public LocalDate getFechaAlta() { return fechaAlta; }
    public void setFechaAlta(LocalDate fechaAlta) { this.fechaAlta = fechaAlta; }
}

```

Si se desea que la clase *Producto* sea un record en lugar de una clase estándar, y al mismo tiempo que se pueda reutilizar el código existente en *ProductoCRUD*, la estructura del record debe mantenerse compatible con los nombres de los campos que ya utiliza *MongoDB* (*id*, *idProducto*, *nombre*, etc.).

```

import org.bson.codecs.pojo.annotations.BsonId;
import org.bson.codecs.pojo.annotations.BsonProperty;
import org.bson.types.ObjectId;
import java.time.LocalDate;

public record Producto(
    @BsonId
    ObjectId id, // ObjectId para MongoDB
    @BsonProperty("idProducto")
    int idProducto, // ID entero para identificar el producto

```

```

    @BsonProperty("nombre")
    String nombre,                // Nombre del producto
    @BsonProperty("precio")
    double precio,                // Precio del producto
    @BsonProperty("stock")
    int stock,                    // Cantidad en stock
    @BsonProperty("fechaAlta")
    LocalDate fechaAlta           // Fecha de alta del producto
) {}

```

➤ ProductoCRUD.java

```

import com.mongodb.MongoClientSettings;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import org.bson.codecs.configuration.CodecRegistry;
import org.bson.codecs.configuration.CodecRegistries;
import org.bson.codecs.pojo.PojoCodecProvider;
import org.bson.Document;
import java.time.LocalDate;
import java.util.List;
import java.util.Scanner;

public class ProductoCRUD {
    private static MongoDatabase database;
    private static MongoCollection<Producto> collection;
    private static final Scanner scanner = new Scanner(System.in);

    public static void conectar() {
        CodecRegistry codecRegistry = CodecRegistries.fromRegistries(
            MongoClientSettings.getDefaultCodecRegistry(),
            CodecRegistries.fromProviders(PojoCodecProvider.builder().automatic(true).build())
        );
        MongoClient mongoClient = MongoClients.create("mongodb://root:root@localhost:27017");
        database = mongoClient.getDatabase("tienda").withCodecRegistry(codecRegistry);
        collection = database.getCollection("productos", Producto.class);
    }

    public static void crearProducto() {
        System.out.println("Introduce los detalles del producto:");
        System.out.print("ID Producto: ");
        int idProducto = scanner.nextInt();
        scanner.nextLine();
        System.out.print("Nombre: ");
        String nombre = scanner.nextLine();
        System.out.print("Precio: ");
        double precio = scanner.nextDouble();
        System.out.print("Stock: ");
        int stock = scanner.nextInt();
        scanner.nextLine();
        System.out.print("Fecha de alta (formato YYYY-MM-DD): ");
        String fechaAltaStr = scanner.nextLine();
        LocalDate fechaAlta = LocalDate.parse(fechaAltaStr);

        Producto producto = new Producto(idProducto, nombre, precio, stock, fechaAlta);
        collection.insertOne(producto);
        System.out.println("Producto creado correctamente.");
    }

    public static void mostrarProducto() {
        System.out.print("Introduce el ID del producto a mostrar: ");
        int idProducto = scanner.nextInt();
        Producto producto = collection.find(new Document("idProducto", idProducto)).first();

        if (producto != null) {
            System.out.println("Producto encontrado:");
            System.out.println("ID Producto: " + producto.getIdProducto());
            System.out.println("Nombre: " + producto.getNombre());
            System.out.println("Precio: " + producto.getPrecio());
            System.out.println("Stock: " + producto.getStock());
            System.out.println("Fecha Alta: " + producto.getFechaAlta());
        } else {
            System.out.println("Producto no encontrado.");
        }
    }
}

```

```

public static void actualizarProducto() {
    System.out.print("Introduce el ID del producto a actualizar: ");
    int idProducto = scanner.nextInt();
    Producto productoExistente = collection.find(new Document("idProducto", idProducto)).first();
    if (productoExistente != null) {
        System.out.print("Nuevo nombre: ");
        String nombre = scanner.next();
        System.out.print("Nuevo precio: ");
        double precio = scanner.nextDouble();
        System.out.print("Nuevo stock: ");
        int stock = scanner.nextInt();
        System.out.print("Nueva fecha alta (YYYY-MM-DD): ");
        String fechaAltaStr = scanner.next();
        LocalDate fechaAlta = LocalDate.parse(fechaAltaStr);

        Producto productoActualizado = new Producto(idProducto, nombre, precio, stock, fechaAlta);
        collection.replaceOne(new Document("idProducto", idProducto), productoActualizado);
        System.out.println("Producto actualizado correctamente.");
    } else {
        System.out.println("Producto no encontrado.");
    }
}

public static void eliminarProducto() {
    System.out.print("Introduce el ID del producto a eliminar: ");
    int idProducto = scanner.nextInt();
    Producto productoExistente = collection.find(new Document("idProducto", idProducto)).first();
    if (productoExistente != null) {
        collection.deleteOne(new Document("idProducto", idProducto));
        System.out.println("Producto eliminado correctamente.");
    } else {
        System.out.println("Producto no encontrado.");
    }
}

public static void mostrarTodosLosProductos() {
    List<Producto> productos = collection.find().into(new java.util.ArrayList<>());
    if (!productos.isEmpty()) {
        System.out.println("Listado de productos:");
        for (Producto producto : productos) {
            System.out.println("ID Producto: " + producto.getIdProducto());
            System.out.println("Nombre: " + producto.getNombre());
            System.out.println("Precio: " + producto.getPrecio());
            System.out.println("Stock: " + producto.getStock());
            System.out.println("Fecha Alta: " + producto.getFechaAlta());
            System.out.println("-----");
        }
    } else {
        System.out.println("No hay productos en la BD.");
    }
}

public static void mostrarMenu() {
    while (true) {
        System.out.println("\n┌─── Menú CRUD de productos ──┐");
        System.out.println("1. Crear producto.           │");
        System.out.println("2. Mostrar producto.        │");
        System.out.println("3. Actualizar producto.     │");
        System.out.println("4. Eliminar producto.       │");
        System.out.println("5. Mostrar todos los productos. │");
        System.out.println("0. Salir.                   │");
        System.out.println("└──────────────────────────┘");
        System.out.print("Selecciona una opción: ");
        int opcion = scanner.nextInt();
        scanner.nextLine();

        switch (opcion) {
            case 1 -> crearProducto();
            case 2 -> mostrarProducto();
            case 3 -> actualizarProducto();
            case 4 -> eliminarProducto();
            case 5 -> mostrarTodosLosProductos();
            case 0 -> {
                System.out.println("Saliendo del programa.");
                return;
            }
        }
    }
}

```

```

        default -> System.out.println("Opción no válida.");
    }
}

public static void main(String[] args) {
    conectar();
    mostrarMenu();
}
}

```



TAREA

4. Crear una aplicación *Java* para gestionar clientes en una *BD MongoDB* con las operaciones *CRUD*. La clase *Cliente* debe tener los siguientes campos: *id* (String), *nombre* (String), *email* (String), *telefono* (String) y *fechaRegistro* (LocalDate). Menú que debe mostrar la aplicación:

Menú CRUD de clientes

1. Crear cliente.
2. Mostrar cliente.
3. Actualizar cliente.
4. Eliminar cliente.
5. Mostrar todos los clientes.
0. Salir.

Seleccione una opción:

5. A partir del ejercicio anterior, donde se implementó un *CRUD* para gestionar clientes en una *BD MongoDB* utilizando una clase estándar, realizar los mismos pasos, pero utilizando un record en lugar de una clase estándar.
6. Crear una aplicación *Java* en entorno gráfico que permita conectar con la *BD* creada en el ejercicio 1 de forma que se pueda mostrar, en una tabla, la información de los juegos introducidos.
- a) Añadir a la aplicación la posibilidad de registrar nuevos juegos, considerando el título como campo obligatorio.
 - b) Añadir a la funcionalidad de registrar nuevos juegos un control para que no sea posible registrar dos juegos con el mismo nombre. En ese caso se mostrará un mensaje de error al usuario y tendrá que proporcionar otro nombre diferente.
 - c) Añadir una nueva funcionalidad que permita eliminar todos los juegos de un mismo género. Preparar un combo que liste los géneros, seleccionando uno de ellos, se podrán eliminar todos los juegos que pertenezcan al mismo.
 - d) Añadir una funcionalidad que permita modificar los datos de un juego.
7. Crear una aplicación *Java* en entorno gráfico que conecte con una *BD NoSQL (MongoDB)*, según los requisitos que se enumeran a continuación:
- a) La *BD* contendrá, al menos, 2 colecciones relacionadas entre ellas. Utilizar alguna colección que contenga estructuras complejas (arrays, datos estructurados, ...).
 - b) Se podrá llevar a cabo el alta de documentos.
 - c) Se podrá llevar a cabo la modificación de documentos.
 - d) Se podrá llevar a cabo la baja de documentos.
 - e) Se podrán llevar a cabo búsquedas simples (por un campo) y complejas (utilizando condiciones para más de un campo).
 - f) Implementar un mecanismo de usuario/contraseña para acceder a la aplicación. Agregar la colección correspondiente para guardar los datos de los usuarios.
 - g) Permitir importar datos desde una *BD* relacional (*MariaDB*, por ejemplo) a alguna colección de la aplicación.
 - h) Permitir exportar datos a alguna tabla de una *BD* relacional (*MariaDB*, por ejemplo).
 - i) Permitir exportar datos de colecciones como ficheros *CSV*.
 - j) Permitir importar datos de colecciones desde ficheros *CSV*.