Unidad 3. Acceso a Bases de Datos documentales

Revisiones

Revisión	Fecha	Descripción
1.0	31-10-2025	Adaptación de los materiales a markdown
1.1	28-11-2025	Modificación de funciones de importación y exportación

3.1. Introducción

Las bases de datos documentales nativas (como MongoDB, Redis o Firebase) almacenan información en forma de documentos, usualmente codificados en JSON, BSON o XML, en lugar de filas y columnas como en las bases de datos relacionales.

Cada documento puede tener una estructura diferente, lo que permite mayor flexibilidad y agilidad en el desarrollo.

Sin embargo, si el dominio de la aplicación tiene muchas relaciones fuertes entre entidades y se necesita garantizar una integridad referencial estricta, una base de datos relacional puede ser más adecuada.

Ventajas

Ventaja	Descripción
Flexibilidad del esquema	No es necesario definir un esquema fijo antes de insertar datos. Ideal para estructuras dinámicas.
Escalabilidad horizontal	Se adaptan bien al escalado distribuyendo los datos en múltiples servidores (sharding).
Rendimiento en lectura y escritura	Muy eficiente en operaciones de lectura y escritura sobre documentos completos.

Ventaja	Descripción
Modelo cercano a objetos	Almacenan los datos de manera similar a como se manejan en el código (objetos serializados como JSON).
Facilidad de integración con APIs REST	Los documentos JSON pueden ser enviados y recibidos fácilmente a través de APIs.
Ideal para datos semiestructurados	Útiles para trabajar con datos que no se ajustan a una estructura tabular, como respuestas de formularios, logs, etc.

Inconvenientes

Inconveniente	Descripción
Falta de integridad referencial	No hay claves foráneas como en las bases de datos relacionales, lo que puede causar inconsistencias si no se gestiona adecuadamente desde la aplicación.
Redundancia de datos	Se repite información entre documentos al no haber normalización; esto puede generar más uso de espacio.
Curva de aprendizaje	Requiere aprender nuevos conceptos como agregaciones, operadores específicos y estructuras de documentos.
Menor soporte para transacciones complejas	Aunque existen transacciones en algunas bases (como MongoDB), su uso es más limitado que en sistemas relacionales.
Consultas menos optimizadas en relaciones complejas	No es la mejor opción cuando los datos necesitan muchas relaciones y joins complejos.

Estructuras básicas de almacenamiento de información

Concepto	Equivalente en BD relacional	Descripción
Base de datos	Base de datos	Conjunto de colecciones.
Colección	Tabla	Agrupación de documentos relacionados.
Documento	Fila (registro)	Unidad básica de almacenamiento. Es un objeto JSON.
Campo	Columna	Atributo dentro del documento.

3.2. JSON

JSON (JavaScript Object Notation) es un formato de texto ligero utilizado para almacenar e intercambiar información estructurada entre aplicaciones. Aunque su sintaxis proviene de JavaScript, hoy en día es independiente del lenguaje y se usa ampliamente en entornos como Kotlin, Java, Python, Node.js, bases de datos NoSQL, APIs REST, etc.

Un fichero JSON está compuesto por **pares clave–valor**, donde:

- Las **claves** siempre van entre comillas dobles " ".
- Los **valores** pueden ser:
- Cadenas de texto ("texto")
- Números (42)
- Booleanos (true o false)
- Objetos (otro conjunto de pares clave-valor { ... })
- Arrays o listas ([...])
- Valor nulo (null)

Ejemplos de estructuras JSON

Objeto simple: Representa un único elemento con propiedades básicas.

```
{
"nombre": "Pol",
"edad": 21,
"ciudad": "Castellón"
}
```

Objeto con array(lista de valores): Incluye un campo que contiene una lista.

```
{
"nombre": "Pol",
"aficiones": ["libros", "cine", "música"]
}
```

Objeto con otro objeto anidado: Un campo puede contener a su vez otro objeto JSON.

```
{
"nombre": "Pol",
"edad": 21,
"direccion": {
    "calle": "Mayor",
    "ciudad": "Castellón",
    "codigo_postal": 12001
}
}
```

Array de objetos: Cuando necesitamos almacenar varios elementos similares (por ejemplo, una lista de productos o alumnos).

```
{
"alumnos": [
    { "nombre": "Pol", "nota": 7.6 },
    { "nombre": "Eli", "nota": 8.2 },
    { "nombre": "Mar", "nota": 9.8 }
]
}
```

Combinación compleja (objetos + arrays + anidamientos): Para representar datos estructurados, como los de una tienda online.

```
{
"pedido": {
    "id": 101,
    "fecha": "2025-10-11",
    "cliente": {
    "nombre": "Pol",
    "email": "pol@dominio.com"
    },
    "productos": [
    { "nombre": "Ensalada de piña", "precio": 10.50, "cantidad": 1 },
    { "nombre": "Tarta de manzana", "precio": 3.50, "cantidad": 1 }
    ],
    "total": 14.00
}
}
```

Array de objetos principales: También se puede usar un array como estructura raíz, por ejemplo, para representar varios registros en un mismo fichero:

```
[
    { "nombre": "Pol", "edad": 21 },
    { "nombre": "Eli", "edad": 22 },
    { "nombre": "Mar", "edad": 18 }
]
```

Práctica 1: Crea y valida tu JSON

1. Crea un fichero `datos.json` con la información con la que estás trabajando.
2. Asegúrate de incluir diferentes tipos de datos (texto, número, booleano, array y objeto anidado).
3. Valida el archivo utilizando https://jsonlint.com

3.3. MongoDB

Introducción

MongoDB es un sistema de gestión de bases de datos NoSQL **orientado a documentos**. A diferencia de las bases de datos relacionales, que almacenan la información en tablas con filas y columnas, MongoDB guarda los datos en **colecciones** formadas por documentos en formato **BSON** (una representación binaria de JSON).

En **MongoDB** cada documento es una **estructura flexible**, parecida a un objeto de programación, donde los datos se organizan en pares **clave–valor**. Esta flexibilidad permite que cada **documento** tenga una estructura diferente, lo que hace que MongoDB se adapte fácilmente a los cambios en los datos sin necesidad de modificar esquemas.

Ejemplo 1: Plantas y jardineros

A continuación tenemos información sobre algunas **plantas** y los **jardineros** que las cuidan. Dependiendo de cómo se deba acceder a la información, se pueden guardar las plantas con sus jardineros, o los jardineros con las plantas que cuidan.

De la primera manera podríamos tener una colección llamada **Plantas**. Observa cómo los objetos no tienen por qué tener la misma estructura y, en este caso, la forma de acceder al nombre de un jardinero sería la siguiente: **objeto.jardinero.nombre**:

```
    {
        "id_planta": 301,
        "nombre": "Rosa silvestre",
        "tipo": "Arbust",
        "jardinero": {
            "nombre": "Eli",
            "apellidos": "Martínez Serra",
            "anyo_nacimiento": 1985
```

```
        },
        "localitzacio": "Jardí Mediterrani"
    },
    {
        "id_planta": 302,
        "nombre": "Ficus lyrata",
        "tipo": "Planta de interior",
        "jardinero": {
            "nombre": "Pol",
            "apellidos": "Ribas Colomer",
            "pais": "Espanya"
        },
        "altura": 150,
        "riego_semanal": 2
    }
```

De la segunda manera tendríamos la colección **Jardineros** donde la información estaría organizasa por jardineros y cada uno de ellos tendría un array con las plantas que cuida (los corchetes: [ ]):

```
    {
        "id_jardinero": 401,
        "nombre": "Eli",
        "apellidos": "Martínez Serra",
        "anyo_nacimiento": 1985,
        "plantas": [
            {
                "nombre": "Rosa silvestre",
                "tipo": "Arbust",
                "ubicacion": "Jardí Mediterrani"
            },
            {
                "nombre": "Lavanda officinalis",
                "ubicacion": "Parterre aromàtic"
            }
        ]
    },
    {
        "id_jardinero": 402,
        "nombre": "Pol",
        "apellidos": "Ribas Colomer",
        "pais": "España",
        "plantas": [
            {
                "nombre": "Ficus lyrata",
                "tipo": "Planta d'interior",
                "altura": 150,
                "riego_semanal": 2
            }
        ]
    }
```

> ⚠ **Práctica 2: Amplía tu JSON**
>
> 1. Amplía el JSON de la práctica anterior para que contenga información estructurada como la del ejemplo anterior (utilizando uno de los dos ejemplos).
> 2. Valida el archivo utilizando https://jsonlint.com

Trabajar con MongoDB

MongoDB utiliza su propia **shell interactiva**, llamada `mongosh`, que permite ejecutar comandos para administrar bases de datos, colecciones y documentos. Su sintaxis es **muy similar a JavaScript**, ya que cada comando se ejecuta sobre un **objeto base**:

```
db.coleccion.operacion()
```

- db → representa la base de datos actual.
- coleccion → el nombre de la colección sobre la que actuamos.
- operacion() → el comando que deseamos ejecutar.

En cualquier operación, debemos escribir db seguido del nombre de la colección y después la operación a realizar. Para guardar un documento ejecutamos el siguiente comando:

```
db.ejemplo.insertOne({ msg: "Hola, ¿qué tal?" })
```

Obtendremos una respuesta indicando que se ha insertado un documento en la **colección ejemplo** (si no existía, la creará automáticamente):

```
    {
    acknowledged: true,
    insertedId: ObjectId('68ff6004ab24a06f35cebea4')
    }
```

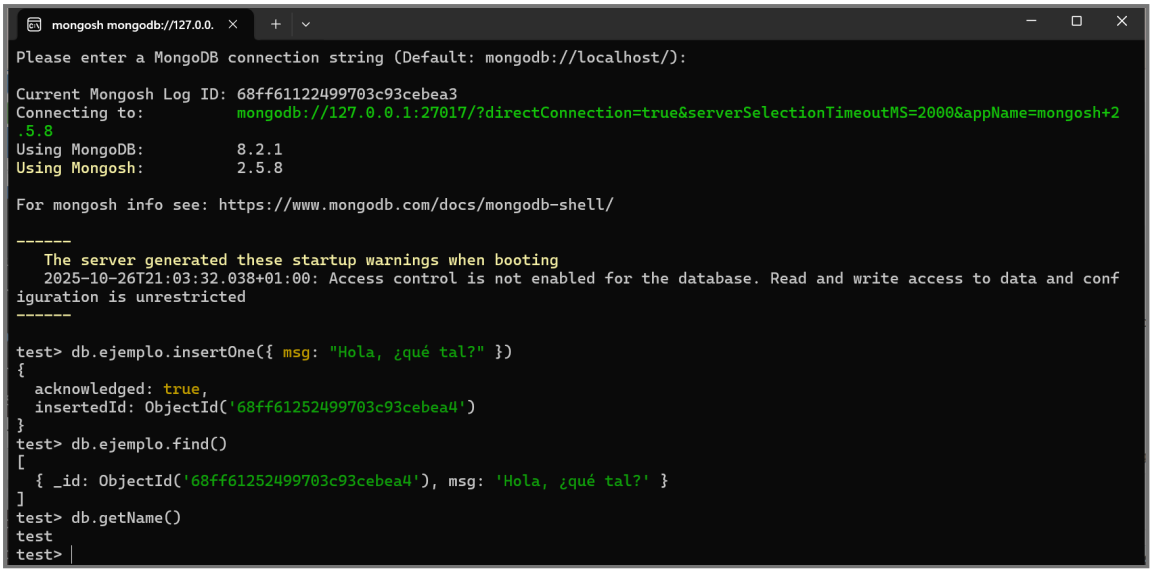
Y con el siguiente comando recuperamos la información:

```
db.ejemplo.find()
```

Lo que nos devolverá algo como:

```
{ "_id" : ObjectId("56cc1acd73b559230de8f71b"), "msg" : "Hola, ¿qué tal?" }
```

Todo esto se realiza en la misma terminal, y cada uno de nosotros obtendrá un número diferente en el campo **ObjectId**. En la siguiente imagen pueden verse las dos operaciones.



Información útil del entorno

Comando	Descripción
`db.stats()`	Muestra estadísticas sobre la base de datos. **Ejemplo:** `db.stats()`
`db.coleccion.stats()`	Muestra estadísticas sobre una colección. **Ejemplo:** `db.alumnos.stats()`
`db.version()`	Devuelve la versión de MongoDB. **Ejemplo:** `db.version()`

Comandos sobre bases de datos

Comando	Descripción	Ejemplo
`show dbs`	Muestra todas las bases de datos existentes.	`show dbs`
`use <nombre>`	Cambia a una base de datos (la crea si no existe).	`use biblioteca`
`db.getName()`	Muestra el nombre de la base de datos actual.	`db.getName()`

Comando	Descripción	Ejemplo
`db.dropDatabase()`	Elimina la base de datos actual.	`db.dropDatabase()`

Comandos sobre colecciones

Comando	Descripción
`show collections`	Lista todas las colecciones de la base de datos. **Ejemplo:** `show collections`
`db.createCollection("nombre")`	Crea una colección vacía. **Ejemplo:** `db.createCollection("alumnos")`
`db.coleccion.drop()`	Elimina una colección completa. **Ejemplo:** `db.alumnos.drop()`
`db.coleccion.renameCollection("nuevoNombre")`	Cambia el nombre de una colección. **Ejemplo:** `db.alumnos.renameCollection("estudiantes")`

Práctica 3: Instala MongoDB

Instala MongoDB en tu ordenador siguiendo la guía Instalación y administración de MongoDB.

Ejemplo 2: Crear BD, insertar plantas y mostrarlas

El siguiente ejemplo crea una base de datos llamada `florabotanica`. Crea una colección llamada `plantas` e inserta tres documentos con campos: `nombre_comun`, `nombre_cientifico`, `altura`. Por último muestra todas las bases de datos y las colecciones creadas.

```
// Abrir mongosh
mongosh

// Crear/usar la base de datos
use florabotanica

// Insertar documentos (si la colección no existe, se crea automáticamente)
```

```
db.plantas.insertMany([
  { id_planta: 1, nombre_comun: "Aloe", nombre_cientifico: "Aloe vera",
altura: 30 },
  { id_planta: 2, nombre_comun: "Pino", nombre_cientifico: "Pinus sylvestris",
altura: 330 },
  { id_planta: 3, nombre_comun: "Cactus", nombre_cientifico: "Cactaceae",
altura: 120 }
])

// Comprobar bases de datos y colecciones
show dbs
show collections
```

El ejemplo funciona de la siguiente manera:

- `use florabotanica` cambia el contexto
- `insertMany` inserta varios documentos
- `show dbs` no mostrará `florabotanica` hasta que la colección tenga datos persistidos;
- tras insertar, aparecerá en la lista

Salida esperada:

```
{ acknowledged: true, insertedIds: { '0': ObjectId("..."), '1':
ObjectId("..."), '2': ObjectId("...") } }

> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
florabotanica 0.001GB

> show collections
plantas
```

Prueba y analiza el ejemplo 2

Prueba el código de ejemplo y verifica que funciona correctamente.

Práctica 4: Crea tu BD, inserta y muestra información

1. Abre la terminal (`mongosh`) y crea tu BD.
2. Crea una colección e inserta tres documentos con los campos que quieras.
3. Muestra todas las bases de datos y las colecciones creadas.

Una vez comprendido el manejo desde terminal, trabajaremos con kotlin a través del *driver oficial de MongoDB para Kotlin*. Para ello crearemos un nuevo proyecto en IntelliJ con Gradle.

Además, para los ejemplos realizados en `Kotlin` de esta unidad, se han declarado tres constantes para almacenar el servidor, el nombre de la BD y el nombre de la colección con los que vamos a trabajar. También se crea una variable Scanner de forma global para poder utilizarla en cualquier parte del programa.

```
const val NOM_SRV = "mongodb://localhost:27017"
const val NOM_BD = "florabotanica"
const val NOM_COLECCION = "plantas"

// Creamos el Scanner de forma global
val scanner = Scanner(System.`in`)
```

Ejemplo 3: Conexión y lectura de información en Kotlin

El siguiente ejemplo añade la dependencia del driver de MongoDB, conecta a la BD `florabotanica` y muestra por consola la información de cada documento JSON almacenado en `plantas`.

1. Añadir dependencia al fichero `build.gradle.kts`

```
    implementation("org.mongodb:mongodb-driver-sync:4.11.0")
```

2. Conectar a la BD y leer la información

```
import com.mongodb.client.MongoClients

fun mostrarPlantas() {
    val cliente = MongoClients.create(NOM_SRV)
    val db = cliente.getDatabase(NOM_BD)
    val coleccion = db.getCollection(NOM_COLECCION)

    // Mostrar documentos de la colección plantas
    val cursor = coleccion.find().iterator()
    cursor.use {
        while (it.hasNext()) {
            val doc = it.next()
            println(doc.toJson())
        }
    }

    cliente.close()
}
```

Prueba y analiza el ejemplo 3

Prueba el código de ejemplo y verifica que funciona correctamente.

Práctica 5: Trabaja con tu BD

1. Crea un proyecto Kotlin en IntelliJ IDEA.
2. Añade la dependencia del driver de MongoDB (`org.mongodb:mongodb-driver-sync`).
3. Conéctate a tu base de datos y muestra los documentos de tu colección. Intenta formatear la información que sale por consola. Por ejemplo:

```
**** Listado de plantas:
[1] Aloe (Aloe vera): 30 cm
[2] Pino (Pinus sylvestris): 330 cm
[3] Cactus (Cactaceae): 120 cm
```

Operaciones básicas

Inserción (Si la colección no existe, MongoDB la **creará automáticamente** en el momento de la inserción)

Comando	Descripción
`insertOne()`	Inserta un solo documento. **Ejemplo:** `db.alumnos.insertOne({nombre:"Ana", nota:8})`
`insertMany()`	Inserta varios documentos a la vez. **Ejemplo:** `db.alumnos.insertMany([{nombre:"Luis", nota:7}, {nombre:"Marta", nota:9}])`

Búsqueda

Comando	Descripción
`find()`	Devuelve todos los documentos de la colección. **Ejemplo:** `db.alumnos.find()`
`findOne()`	Devuelve el primer documento que cumple una condición. **Ejemplo:** `db.alumnos.findOne({nombre:"Ana"})`
`find(criterio, proyección)`	Permite filtrar y mostrar solo algunos campos. **Ejemplo:** `db.alumnos.find({nota:{$gte:8}}, {nombre:1, _id:0})`

Operadores comunes:
`$eq` (igual), `$ne` (distinto), `$gt` (mayor que), `$lt` (menor que), `$gte` (mayor o igual), `$lte` (menor o igual), `$in`, `$and`, `$or`.

Actualización (Usa `$set` para modificar solo algunos campos y **no perder el resto**)

Comando	Descripción
`updateOne(filtro, cambios)`	Actualiza el primer documento que cumpla la condición. **Ejemplo:** `db.alumnos.updateOne({nombre:"Ana"}, {$set:{nota:9}})`
`updateMany(filtro, cambios)`	Actualiza todos los documentos que cumplan la condición. **Ejemplo:** `db.alumnos.updateMany({nota:{$lt:5}}, {$set:{aprobado:false}})`
`replaceOne(filtro, nuevoDoc)`	Sustituye el documento completo. **Ejemplo:** `db.alumnos.replaceOne({nombre:"Ana"}, {nombre:"Ana", nota:10})`

Eliminación

Comando	Descripción
`deleteOne()`	Elimina el primer documento que cumpla la condición. **Ejemplo:** `db.alumnos.deleteOne({nombre:"Luis"})`
`deleteMany()`	Elimina todos los documentos que cumplan la condición. **Ejemplo:** `db.alumnos.deleteMany({nota:{$lt:5}})`

Ejemplo 4: Operaciones CRUD en terminal

El siguiente ejemplo realiza las siguientes operaciones sobre la colección `plantas`:

1. Inserta tres nuevos documentos con `insertMany()`.
2. Recupera todos los documentos con `find()`.
3. Filtra aquellos cuya `altura` sea mayor de 100.
4. Actualiza uno de los documentos cambiando la altura.
5. Elimina una planta específica mediante `deleteOne()`.

```
// 1) Insertar tres nuevos documentos
db.plantas.insertMany([
  { id_planta: 4, nombre_comun: "Lavanda", nombre_cientifico: "Lavandula",
altura: 50, tipo: "arbusto" },
  { id_planta: 5, nombre_comun: "Rosal", nombre_cientifico: "Rosa", altura:
120, tipo: "arbusto" },
  { id_planta: 6, nombre_comun: "Olivo", nombre_cientifico: "Olea europaea",
altura: 800, tipo: "árbol" }
])

// 2) Recuperar todos los documentos
db.plantas.find().pretty()

// 3) Filtrar altura > 100
db.plantas.find({ altura: { $gt: 100 } }).pretty()

// 4) Actualizar: cambiar altura de "Cactus" a 130
db.plantas.updateOne({ nombre_comun: "Cactus" }, { $set: { altura: 130 } })

// 5) Eliminar una planta por nombre
db.plantas.deleteOne({ nombre_comun: "Rosal" })
```

El ejemplo funciona de la siguiente manera:

- `insertMany` devuelve `acknowledged: true` con `insertedIds`.
- `find().pretty()` muestra documentos en JSON formateado.
- `find({ altura: { $gt: 100 }})` listará pinos, olivos, etc.
- `updateOne` devuelve un objeto con `matchedCount` y `modifiedCount`.
- `deleteOne` devuelve `deletedCount: 1` si eliminó un documento.

Salida de `updateOne`:

```
{ acknowledged: true, matchedCount: 1, modifiedCount: 1, upsertedId: null }
```

Salida de `deleteOne`:

```
{ acknowledged: true, deletedCount: 1 }
```

Prueba y analiza el ejemplo 4

Prueba el código de ejemplo y verifica que funciona correctamente.

> **Práctica 6: Trabaja con tu BD**
>
> 1. Inserta tres nuevos documentos con `insertMany()`.
> 2. Recupera todos los documentos con `find()`.
> 3. Aplica algún filtro.
> 4. Actualiza uno de los documentos cambiando el valor de un campo.
> 5. Elimina un documento específico mediante `deleteOne()`.

Ejemplo 5: Operaciones CRUD desde Kotlin

El siguiente fragmento de código realiza las siguientes operaciones sobre la colección `plantas` de la BD `florabotanica`:

1. Insertar un nuevo documento a partir de los datos introducidos por el usuario.
2. Actualizar la altura de una planta dada.
3. Eliminar una planta por nombre.

```kotlin
import com.mongodb.client.MongoClients
import com.mongodb.client.MongoDatabase
import com.mongodb.client.model.Filters
import org.bson.Document
import java.util.Scanner

fun insertarPlanta() {
    //conectar con la BD
    val cliente = MongoClients.create(NOM_SRV)
    val db = cliente.getDatabase(NOM_BD)
    val coleccion = db.getCollection(NOM_COLECCION)

    var id_planta: Int? = null
    while (id_planta == null) {
        print("ID de la planta: ")
        val entrada = scanner.nextLine()
        id_planta = entrada.toIntOrNull()
        if (id_planta == null) {
            println("El ID debe ser un número !!!")
        }
    }

    print("Nombre común: ")
    val nombre_comun = scanner.nextLine()
    print("Nombre científico: ")
    val nombre_cientifico = scanner.nextLine()

    var altura: Int? = null
    while (altura == null) {
        print("Altura (en cm): ")
        val entrada = scanner.nextLine()
        altura = entrada.toIntOrNull()
```

```
        if (altura == null) {
            println("¡¡¡ La altura debe ser un número !!!")
        }
    }

    val doc = Document("id_planta", id_planta)
        .append("nombre_comun", nombre_comun)
        .append("nombre_cientifico", nombre_cientifico)
        .append("altura", altura)

    coleccion.insertOne(doc)
    println("Planta insertada con ID: ${doc.getObjectId("_id")}")

    cliente.close()
    println("Conexión cerrada")
}


fun actualizarAltura() {
    //conectar con la BD
    val cliente = MongoClients.create(NOM_SRV)
    val db = cliente.getDatabase(NOM_BD)
    val coleccion = db.getCollection(NOM_COLECCION)

    var id_planta: Int? = null
    while (id_planta == null) {
        print("ID de la planta a actualizar: ")
        val entrada = scanner.nextLine()
        id_planta = entrada.toIntOrNull()
        if (id_planta == null) {
            println("El ID debe ser un número !!!")
        }
    }

    //comprobar si existe una planta con el id_planta proporcionado por
consola
    val planta = coleccion.find(Filters.eq("id_planta",
id_planta)).firstOrNull()
    if (planta == null) {
        println("No se encontró ninguna planta con id_planta =
\"$id_planta\".")
    }
    else {
        // Mostrar información de la planta encontrada
        println("Planta encontrada: ${planta.getString("nombre_comun")}
(altura: ${planta.get("altura")} cm)")

        //pedir nueva altura
        var altura: Int? = null
        while (altura == null) {
            print("Nueva altura (en cm): ")
            val entrada = scanner.nextLine()
            altura = entrada.toIntOrNull()
            if (altura == null) {
                println("¡¡¡ La altura debe ser un número !!!")
            }
        }
```

```
        // Actualizar el documento
        val result = coleccion.updateOne(
            Filters.eq("id_planta", id_planta),
            Document("\$set", Document("altura", altura))
        )

        if (result.modifiedCount > 0)
            println("Altura actualizada correctamente (${result.modifiedCount}
documento modificado).")
        else
            println("No se modificó ningún documento (la altura quizá ya era
la misma).")
    }

    cliente.close()
    println("Conexión cerrada.")
}


fun eliminarPlanta() {
    //conectar con la BD
    val cliente = MongoClients.create(NOM_SRV)
    val db = cliente.getDatabase(NOM_BD)
    val coleccion = db.getCollection(NOM_COLECCION)

    var id_planta: Int? = null
    while (id_planta == null) {
        print("ID de la planta a eliminar: ")
        val entrada = scanner.nextLine()
        id_planta = entrada.toIntOrNull()
        if (id_planta == null) {
            println("El ID debe ser un número !!!")
        }
    }

    val result = coleccion.deleteOne(Filters.eq("id_planta", id_planta))
    if (result.deletedCount > 0)
        println("Planta eliminada correctamente.")
    else
        println("No se encontró ninguna planta con ese nombre.")

    cliente.close()
    println("Conexión cerrada.")
}
```

Prueba y analiza el ejemplo 5

Prueba el código de ejemplo y verifica que funciona correctamente.

Práctica 7: Trabaja con tu BD

Amplía tu proyecto con las funciones para las operaciones CRUD y un menú con las siguientes opciones:

1. Listar todos los documentos existentes.
2. Insertar un nuevo documento (a partir de los datos introducidos por consola).
3. Actualizar la informaión de un documento (por ID).
4. Eliminar un documento (por ID).

Consultas avanzadas y ordenación

Comando	Descripción
`sort()`	Ordena los resultados. `1` ascendente, `-1` descendente. **Ejemplo:** `db.alumnos.find().sort({nota:-1})`
`limit()`	Limita el número de resultados. **Ejemplo:** `db.alumnos.find().limit(3)`
`countDocuments()`	Devuelve el número de documentos que cumplen un filtro. **Ejemplo:** `db.alumnos.countDocuments({nota:{$gte:5}})`

Índices

Comando	Descripción
`createIndex({campo:1})`	Crea un índice ascendente. **Ejemplo:** `db.alumnos.createIndex({nombre:1})`
`getIndexes()`	Muestra los índices existentes. **Ejemplo:** `db.alumnos.getIndexes()`
`dropIndex("nombre_1")`	Elimina un índice. **Ejemplo:** `db.alumnos.dropIndex("nombre_1")`

Consultas avanzadas con `aggregate()`

El método `aggregate()` permite realizar **consultas complejas** y **procesamientos de datos** en varias etapas, similares a las funciones de **GROUP BY, JOIN o HAVING** en SQL. Cada etapa del

pipeline (tubería) transforma los datos paso a paso. Cada etapa (stage) se representa mediante un objeto precedido por $, que indica la operación a realizar.

Estructura básica

```
db.coleccion.aggregate([
{ <etapa1> },
{ <etapa2> },
...
])
```

Etapa	Descripción
`$match`	Filtra documentos (equivalente a `WHERE`). **Ejemplo:** `{ $match: { ciudad: "Valencia" } }`
`$project`	Selecciona campos específicos o crea nuevos. **Ejemplo:** `{ $project: { _id:0, nombre:1, nota:1 } }`
`$sort`	Ordena los resultados. **Ejemplo:** `{ $sort: { nota: -1 } }`
`$limit`	Limita el número de resultados. **Ejemplo:** `{ $limit: 5 }`
`$skip`	Omite un número de documentos. **Ejemplo:** `{ $skip: 10 }`
`$group`	Agrupa los documentos por un campo y calcula valores agregados (como `COUNT`, `SUM`, `AVG`). **Ejemplo:** `{ $group: { _id: "$curso", media: { $avg: "$nota" } } }`
`$count`	Devuelve el número total de documentos resultantes. **Ejemplo:** `{ $count: "total" }`
`$lookup`	Realiza una unión entre colecciones (similar a `JOIN`). **Ejemplo:** `{ $lookup: { from: "profesores", localField: "idProfesor", foreignField: "_id", as: "infoProfesor" } }`
`$unwind`	Descompone arrays en múltiples documentos. **Ejemplo:** `{ $unwind: "$aficiones" }`

Ejemplo 6: Consultas avanzadas y agregaciones

El siguiente ejemplo realiza lo siguiente:

1. Usa `aggregate()` para calcular la altura media de las plantas.
2. Agrupa por tipo de planta con `$group` y ordena los resultados.
3. Limita la salida a los tres resultados más altos con `$limit`.

```
// Calcular altura media de todas las plantas
db.plantas.aggregate([
  { $group: { _id: null, alturaMedia: { $avg: "$altura" } } }
])

// Agrupar por tipo y calcular media, ordenar descendente
db.plantas.aggregate([
  { $match: { tipo: { $exists: true } } },
  { $group: { _id: "$tipo", mediaAltura: { $avg: "$altura" }, cantidad: {
$sum: 1 } } },
  { $sort: { mediaAltura: -1 } }
])

// Obtener los 3 más altos
db.plantas.aggregate([
  { $sort: { altura: -1 } },
  { $limit: 3 },
  { $project: { _id:0, nombre_comun:1, altura:1 } }
])
```

Salida esperada:

```
// Resultado del primer aggregate
{ "_id" : null, "alturaMedia" : 250.25 }

// Resultado del group
{ "_id" : "árbol", "mediaAltura" : 540, "cantidad" : 1 }

// Resultado del limit
{ "nombre_comun" : "Olivo", "altura" : 800 }
{ "nombre_comun" : "Pino", "altura" : 330 }
{ "nombre_comun" : "Cactus", "altura" : 130 }
```

Prueba y analiza el ejemplo 6

Prueba el código de ejemplo y verifica que funciona correctamente.

Práctica 8: Trabaja sobre tu BD

1. Usa `aggregate()` para realizar algún cálculo.
2. Agrupa por tipo o categoría utilizando `$group` y ordena los resultados.
3. Limita la salida a los tres resultados más altos con `$limit`.

Ejemplo 7: Consultas avanzadas en Kotlin

El siguiente ejemplo conecta a la BD `florabotanica` y realiza las siguientes operaciones:

1. Implementa consultas utilizando filtros con `Filters.eq`, `Filters.gt`, etc.
2. Muestra solo los nombres de las plantas con `Projections.include`.
3. Realiza una agregación que calcule la media de alturas.

```
import com.mongodb.client.model.Projections

fun variasOperaciones() {
    val client = MongoClients.create(NOM_SRV)
    val col = client.getDatabase(NOM_BD).getCollection(NOM_COLECCION)

    println("*****Plantas que miden más de 100cm")
    // 1) Filtro: altura > 100
    col.find(Filters.gt("altura", 100)).forEach { println(it.toJson()) }

    println("*****Nombre común de todas las plantas")
    // 2) Proyección: solo nombre_comun
    col.find().projection(Projections.include("nombre_comun")).forEach {
println(it.toJson()) }

    println("*****Altura media de todas las plantas")
    // 3) Agregación: media de altura
    val pipeline = listOf(
        Document("\$group", Document("_id", null).append("alturaMedia",
Document("\$avg", "\$altura")))
    )
    val aggCursor = col.aggregate(pipeline).iterator()
    aggCursor.use {
        while (it.hasNext()) println(it.next().toJson())
    }

    client.close()
}
```

Prueba y analiza el ejemplo 7

Prueba el código de ejemplo y verifica que funciona correctamente.

Práctica 9: Trabaja con tu BD

1. Implementa consultas utilizando filtros con `Filters.eq`, `Filters.gt`, etc.
2. Muestra solo algunos datos con `Projections.include`.
3. Realiza una agregación que realice algún cálculo sobre tus datos.

Entrega 1

Realiza lo siguiente:

1. Exporta tu BD a un archivo .json a la carpeta resources (Puedes consultar el apartado `Exportar / Importar la BD con Kotlin` al final de este documento).
2. Entrega en Aules la carpeta `main` de tu proyecto comprimida en formato .zip

IMPORTANTE: El proyecto no debe contener código que no se utilice, ni restos de pruebas de los ejemplos y no debe estar separado por prácticas. Debe ser un proyecto totalmente funcional.

Otras formas de trabajar con BD MongoDB

Además de instalar el servidor en local en nuestro ordenador, podemos utilizar el laboratorio de AWS. Tienes las instrucciones en la guía Instalación y administración de MongoDB.

También podemos trabajar directamente del servidor online `Atlas` que proporciona MongoDB.

Por último se puede trabajar en local sin necesidad de servidor aunque, en este caso, la información no se guarda en disco sino en memoria y, por tanto, al cerrar la aplicación los datos se borran. Esto no es problema si se exportan a un fichero json antes de cerrar el programa y se importan al iniciarlo. Veamos un ejemplo:

Ejemplo 8: Trabajando MongoDB en local (en memoria)

Partimos del fichero `json` de la `Entrega 1`. A continuación se muestra la infromación de la colección `plantas` de la BD `florabotanica`:

```
[
{
  "_id": {
    "$oid": "69160748005c40ac579dc29d"
  },
  "id_planta": 1,
  "nombre_comun": "Aloe",
  "nombre_cientifico": "Aloe barbadensis miller",
  "altura": 60
},
```

```
{
  "_id": {
    "$oid": "69160748005c40ac579dc29e"
  },
  "id_planta": 2,
  "nombre_comun": "Ficus",
  "nombre_cientifico": "Ficus benjamina",
  "altura": 330
},
{
  "_id": {
    "$oid": "69160748005c40ac579dc29f"
  },
  "id_planta": 3,
  "nombre_comun": "Romero",
  "nombre_cientifico": "Rosmarinus officinalis",
  "altura": 120
},
{
  "_id": {
    "$oid": "69160928005c40ac579dc2a0"
  },
  "id_planta": 4,
  "nombre_comun": "Lavanda",
  "nombre_cientifico": "Lavandula angustifolia",
  "altura": 50
},
{
  "_id": {
    "$oid": "69160928005c40ac579dc2a1"
  },
  "id_planta": 5,
  "nombre_comun": "Clavel",
  "nombre_cientifico": "Dianthus caryophyllus",
  "altura": 60
}]
```


El código Kotlin para trabajar con MongoDB en memoria de forma local es el siguiente:

build.gradle.kts

```
dependencies {
    implementation("org.mongodb:mongodb-driver-sync:4.10.2")
    implementation("de.bwaldvogel:mongo-java-server:1.45.0")
    implementation("org.json:json:20231013")

    // Backend de logging para SLF4J
    //implementation("ch.qos.logback:logback-classic:1.5.6")

    //desactivar logs en consola
    implementation("org.slf4j:slf4j-nop:2.0.12")
}
```

Main.kt

```
import java.util.Scanner
import java.io.File
import org.bson.Document
import org.json.JSONArray
import org.bson.json.JsonWriterSettings

import de.bwaldvogel.mongo.MongoServer
import de.bwaldvogel.mongo.backend.memory.MemoryBackend

import com.mongodb.client.MongoClients
import com.mongodb.client.MongoClient
import com.mongodb.client.MongoCollection
import com.mongodb.client.model.Filters
import com.mongodb.client.model.Projections
import com.mongodb.client.model.Aggregates

//variables globales definidas sin inicializar
lateinit var servidor: MongoServer
lateinit var cliente: MongoClient
lateinit var uri: String
lateinit var coleccionPlantas: MongoCollection<Document>

//BD y colección con la que se trabajará
const val NOM_BD = "florabotanica"
const val NOM_COLECCION = "plantas"

// Función para conectar a la BD
fun conectarBD() {
    servidor = MongoServer(MemoryBackend())
    val address = servidor.bind()
    uri = "mongodb://${address.hostName}:${address.port}"

    cliente = MongoClients.create(uri)
    coleccionPlantas =
cliente.getDatabase(NOM_BD).getCollection(NOM_COLECCION)

    println("Servidor MongoDB en memoria iniciado en $uri")
}

// Función para desconectar a la BD
fun desconectarBD() {
    cliente.close()
    servidor.shutdown()
    println("Servidor MongoDB en memoria finalizado")
}

fun main() {
    conectarBD()
    importarBD("src/main/resources/florabotanica_plantas.json",
coleccionPlantas)

    menu()


exportarBD(coleccionPlantas,"src/main/resources/florabotanica_plantas.json")
    desconectarBD()
}
```

```
// ****************
// **** MENÚ   ****
// ****************

fun menu(){
    //llamada a listar todas las plantas de la BD
    mostrarPlantas()
}

fun mostrarPlantas() {
    println();
    println("**** Listado de plantas:")
    coleccionPlantas.find().forEach { doc ->
        val id = doc.getInteger("id_planta")
        val nombre_comun = doc.getString("nombre_comun")
        val nombre_cientifico = doc.getString("nombre_cientifico")
        val altura = doc.getInteger("altura")
        println("[$id] $nombre_comun ($nombre_cientifico): ${altura} cm")
    }
}
```

Hay que tener en cuenta que cuando se trabaja con un servidor en memoria no podemos abrir y cerrar la conexión a la BD en cada operación ya que todo está en la RAM y no en disco. Cada vez que se cierra la conexión y el servidor, la base de datos desaparece y por esta razón la abrimos al iniciar el programa y la cerramos al finalizarlo. También importamos la informaación y la exportamos para tener siempre una copia en formato `json`.

Prueba y analiza el ejemplo 8

Prueba el código del ejemplo y verifica que funciona correctamente.

Práctica 10: Trabaja con tu BD

1. Crea un nuevo proyecto kotlin.
2. Copia el fichero `json` que exportaste en la práctica anterior a la carpeta `resources`.
3. Añade el código de tu anterior proyecto modificando lo que sea necesario para que funcionen todas las opciones de la práctica anterior pero trabajando la BD en memoria.
4. Comprueba que al salir del programa el fichero `json` contiene la información actualizada.

Trabajando con más de una colección

En este punto vamos a profundizar en la utilización de `aggregate()` para poder realizar **consultas complejas** y **procesamientos de datos**. Para ello utilizaremos una lista de etapas (*stages*) que MongoDB ejecutará **en orden** para transformar, combinar o procesar documentos

de una colección. Esa lista la guardaremos en una **tubería de pasos** (*pipeline*), donde la salida de un paso es la entrada del siguiente.

Ya hemos visto que listar el contenido de una colección es muy fácil utilizando `find`, pero si queremos realizar consultas que obtengan datos de varias colecciones hay que realizar operaciones similares al `JOIN` de `SQL`.

Para los ejemplos siguientes añadiremos una nueva colección (`facturas`) a nuestra BD. A continuación se muestra su estructura e información inicial:

Campo	Tipo
fecha	String (formato YYYY-MM-DD)
id_factura	Integer
id_planta	Integer
precio	Integer
cantidad	Integer


```
[
{
"fecha": "2025-11-28",
"id_factura": 1,
"id_planta": 1,
"precio": 13,
"cantidad": 3
},
{
"fecha": "2025-11-28",
"id_factura": 1,
"id_planta": 3,
"precio": 7,
"cantidad": 2
},
{
"fecha": "2025-11-28",
"id_factura": 1,
"id_planta": 5,
"precio": 5,
"cantidad": 1
},
{
"fecha": "2025-11-28",
"id_factura": 2,
"id_planta": 2,
"precio": 35,
```

```
"cantidad": 1
},
{
"fecha": "2025-11-28",
"id_factura": 2,
"id_planta": 4,
"precio": 9,
"cantidad": 2
},
{
"fecha": "2025-11-29",
"id_factura": 3,
"id_planta": 1,
"precio": 13,
"cantidad": 1
},
{
"fecha": "2025-11-29",
"id_factura": 3,
"id_planta": 3,
"precio": 7,
"cantidad": 3
},
{
"fecha": "2025-11-29",
"id_factura": 4,
"id_planta": 2,
"precio": 35,
"cantidad": 2
},
{
"fecha": "2025-11-29",
"id_factura": 4,
"id_planta": 5,
"precio": 5,
"cantidad": 4
}
]
```


Para realizar el JOIN entre las dos colecciones (`plantas` y `facturas`) utilizaremos **lookup y unwind**.

`lookup` añade un nuevo campo que contiene un array con los documentos completos de otra colección cuyo campo coincide con el del documento actual. Incluso si solo encuentra un documento, el resultado sigue siendo un array con un único elemento.

Partimos del primer documento de la colección `facturas`:


```
{
  "fecha": "2025-11-28",
  "id_factura": 1,
  "id_planta": 1,
  "precio": 13,
  "cantidad": 3
}
```

Utilizamos `lookup` para buscar en la colección **plantas** todos los documentos cuyo campo `id_planta` coincida con el `id_planta` de la factura y gurdarlos en un nuevo campo llamado `planta`. El código es el siguiente:

```
Document("\$lookup", Document()
.append("from", "plantas")
.append("localField", "id_planta")
.append("foreignField", "id_planta")
.append("as", "planta")
)
```

El resultado (equivalente a un **JOIN** en SQL) es un campo llamado `planta` añadido al documento:

```
{
  "fecha": "2025-11-28",
  "id_factura": 1,
  "id_planta": 1,
  "precio": 13,
  "cantidad": 3
  "planta": [
    {
      "nombre_comun": "Aloe",
      "nombre_cientifico": "Aloe barbadensis miller",
      "altura": 60,
      "id_planta": 1
    }
  ]
}
```

Para poder leer la información hemos de convertir el resultado del lookup (array) en un objeto normal. Eso es lo que hace `unwind`. Si partimos del array que se ha creado con `lookup`:

```
"planta": [
  { nombre_comun: "Aloe", ... }
]
```

Después de aplicar `unwind` el documento de planta ya no es un `array` y queda como un objeto normal para poder leer sus campos.

```
"planta": {
  nombre_comun: "Aloe",
  ...
}
```

En este caso, el pipeline es una secuencia de dos pasos:

1. `lookup` que junta facturas con plantas (como un JOIN).
2. `unwind` que convierte el resultado del lookup (array) en un objeto normal.

Ejemplo 9: Mostrar listado de factura con el nombre de la planta

Este ejemplo utiliza el `pipeline` explicado anteriormente con la secuencia `lookup` y `unwind` para mostrar el nombre de la planta al listar los documentos de la colección `facturas`.

```
 val pipeline = listOf(
        Document("\$lookup", Document()
            .append("from", "plantas")
            .append("localField", "id_planta")
            .append("foreignField", "id_planta")
            .append("as", "planta")
        ),
        Document("\$unwind", "\$planta")
    )

    coleccionFacturas.aggregate(pipeline).forEach { doc ->
        val idFactura = doc.getInteger("id_factura")
        val fecha = doc.getString("fecha")
        val idPlanta = doc.getInteger("id_planta")
        val cantidad = doc.getInteger("cantidad")
        val precio = doc.getInteger("precio")

        val planta = doc["planta"] as Document
        val nombreComun = planta.getString("nombre_comun")

        println("[$idFactura] ($fecha): $nombreComun (id $idPlanta) -
$cantidad uds. $precio €")
    }
```

Prueba y analiza el ejemplo 9

Prueba el código del ejemplo y verifica que funciona correctamente.

Ejemplo 10: Mostrar datos de una factura

En este ejemplo se pide un número de factura por consola y se muestran sus datos.

```
fun mostrarFactura() {
    val idFactura = pedirEntero("ID de la factura: ")

    // Obtener la fecha de la factura y verificar que la factura indicada
existe
    val facturaDoc = coleccionFacturas
        .find(Document("id_factura", idFactura))
        .first()

    if (facturaDoc == null) {
        println("No existe ninguna factura con ID $idFactura")
        return
    }

    val fecha = facturaDoc["fecha"] as String
```

```
    // Crear un pipeline de agregación para obtener las líneas de la factura
con datos de la planta
    val pipeline = listOf(
        Document("\$match", Document("id_factura", idFactura)),
        Document("\$lookup", Document()
            .append("from", "plantas")
            .append("localField", "id_planta")
            .append("foreignField", "id_planta")
            .append("as", "planta")
        ),
        Document("\$unwind", "\$planta"),
        Document("\$project", Document()
            .append("nombre_planta", "\$planta.nombre_comun")
            .append("cantidad", 1)
            .append("precio", 1)
            .append("subtotal", Document("\$multiply", listOf("\$precio",
"\$cantidad")))
        )
    )

    // Ejecutar la agregación para obtener la lista de líneas
    val lineas = coleccionFacturas.aggregate(pipeline).toList()

    if (lineas.isEmpty()) {
        println("No se encontraron líneas para la factura $idFactura")
        return
    }

    // Encabezado de la factura
    println("======================================================================")
    println("Factura ID: $idFactura")
    println("Fecha: $fecha")
    println("----------------------------------------------------------------------")
    println(String.format("%-15s %-10s %-10s %-12s", "Planta", "Cantidad",
"Precio", "Subtotal"))
    println("----------------------------------------------------------------------")

    var totalFactura = 0.0

    // Iterar sobre las líneas de la factura
    lineas.forEach { linea ->
        val nombre = linea["nombre_planta"] as String
        val cantidad = linea["cantidad"] as Int
        val precio = linea["precio"] as Int
        val subtotal = (linea["subtotal"] as Number).toDouble()

        totalFactura += subtotal

        println(String.format("%-15s %-10d %-10s %-12s",
            nombre, cantidad, precio, subtotal
        ))
    }

    var totalIVA =totalFactura*0.21

    // Mostrar pie de factura con totales
```

```
    println("------------------------------------------------------------")
    println(String.format("%-15s %-10s %-10s %-12s", "", "TOTAL:",
totalFactura, ""))
    println(String.format("%-15s %-10s %-10s %-12s", "", "IVA 21%:", totalIVA,
""))
    println(String.format("%-15s %-10s %-10s %-12s", "", "TOTAL CON IVA:",
totalFactura + totalIVA, ""))
    println("============================================================")
}
```

Prueba y analiza el ejemplo 10

Prueba el código del ejemplo y verifica que funciona correctamente.

Práctica 11: Trabaja con tu BD

1. Añade una nueva colección a tu BD.
2. Añade al menú las operaciones CRUD.
3. Programa una función parecida a la de los ejemplos en la que tengas que realizar una consulta para extraer información de las dos colecciones de tu BD.
4. Recuerda importar y exportar la nueva colección.

Ejemplo 11: Trabajando con una tercera colección

Vamos a completar un poco más nuestra aplicación añadiendo la colección de `Clientes`. Los datos iniciales son los sigueintes:

```
[{
"nombre": "Pol",
"id_cliente": 1
},
{
"nombre": "Ade",
"id_cliente": 2
}]
```

Vamos a practicar realizando las siguientes consultas:

Consulta 1: Mostrar el nombre del cliente en la factura del ejemplo anterior

La cabecera de la función `mostrarFactura` del ejemplo anterior era la siguiente:

```
============================================================
Factura ID: 1
Fecha: 2025-11-28
```

Con la modificación queda así:

```
================================================================
Factura ID: 1
Fecha: 2025-11-28
Cliente: Pol
```

Para obtener el nombre del cliente hay que añadir las siguientes instrucciones:

```
val idCliente = facturaDoc.getInteger("id_cliente")

val clienteDoc = coleccionClientes
    .find(Document("id_cliente", idCliente))
    .first()

val nomCliente = clienteDoc.getString("nombre")
```

Consulta 2: Historial de compras

Para este ejemplo vamos a unir las tres colecciones para mostrar el historial de compras. El resultado (sin formatear) será el siguiente:

```
{"cliente": "Ade", "id_factura": 4, "fecha": "2025-11-29", "planta": "Ficus",
"cantidad": 2, "precio": 35}
{"cliente": "Ade", "id_factura": 4, "fecha": "2025-11-29", "planta": "Clavel",
"cantidad": 4, "precio": 5}
{"cliente": "Pol", "id_factura": 1, "fecha": "2025-11-28", "planta": "Aloe",
"cantidad": 3, "precio": 13}
{"cliente": "Pol", "id_factura": 1, "fecha": "2025-11-28", "planta": "Romero",
"cantidad": 2, "precio": 7}
{"cliente": "Pol", "id_factura": 1, "fecha": "2025-11-28", "planta": "Clavel",
"cantidad": 1, "precio": 5}
{"cliente": "Pol", "id_factura": 2, "fecha": "2025-11-28", "planta": "Ficus",
"cantidad": 1, "precio": 35}
{"cliente": "Pol", "id_factura": 2, "fecha": "2025-11-28", "planta":
"Lavanda", "cantidad": 2, "precio": 9}
{"cliente": "Pol", "id_factura": 3, "fecha": "2025-11-29", "planta": "Aloe",
"cantidad": 1, "precio": 13}
{"cliente": "Pol", "id_factura": 3, "fecha": "2025-11-29", "planta": "Romero",
"cantidad": 3, "precio": 7}
{"cliente": "Pol", "id_factura": 90, "fecha": "2025-12-02", "planta":
"dfdfdfd", "cantidad": 2, "precio": 50}
```

Para obtener el nombre del cliente en cada una de las líneas de factura, habrá que hacer un nuevo **lookup** y un nuevo **unwind** con la colección de clientes, por tanto añadiremos a la 'pipeline' el siguiente código:

```
    Document("\$lookup", Document()
        .append("from", "clientes")
        .append("localField", "id_cliente")
        .append("foreignField", "id_cliente")
        .append("as", "cliente")
```

```
    ),
    Document("\$unwind", "\$cliente"),
```

y la proyección quedará de esta forma:

```
Document("\$project", Document()
    .append("_id", 0)
    .append("id_factura", 1)
    .append("fecha", 1)
    .append("planta", "\$planta.nombre_comun")
    .append("cantidad", 1)
    .append("precio", 1)
    .append("subtotal", Document("\$multiply", listOf("\$precio",
"\$cantidad")))
)
```

Consulta 3: Total gastado por cliente

Esta consulta saca como resultado una línea por cada cliente con su nombre, la suma total de todas las líneas de facturas (precio * cantidad) y el número total de líneas:

```
{"_id": "Ade", "total_gastado": 90, "lineas_compradas": 2}
{"_id": "Pol", "total_gastado": 245, "lineas_compradas": 8}
```

En este caso, los `lookup` y `unwind` serán los mismos que en la consulta anterior pero en vez de una proyección necesitamos un agrupamiento. Este es el código:

```
    Document("\$group", Document()
        .append("_id", "\$cliente.nombre")
        .append("total_gastado", Document("\$sum",
            Document("\$multiply", listOf("\$precio", "\$cantidad"))
        ))
        .append("lineas_compradas", Document("\$sum", 1))
    )
```

Consulta 4: Plantas más vendidas con clientes compradores

En este caso vamos a obtener la información ordenada de las plantas que se han vendido junto a los clientes que las han comprado. El listado (sin formatear) es el sigueinte:

```
{"_id": "Clavel", "total_vendida": 5, "clientes": ["Ade", "Pol"]}
{"_id": "Romero", "total_vendida": 5, "clientes": ["Pol"]}
{"_id": "Aloe", "total_vendida": 4, "clientes": ["Pol"]}
{"_id": "Ficus", "total_vendida": 3, "clientes": ["Ade", "Pol"]}
{"_id": "Lavanda", "total_vendida": 2, "clientes": ["Pol"]}
{"_id": "dfdfdfd", "total_vendida": 2, "clientes": ["Pol"]}
```

Para obtener la información de esta forma el código necesario, además de los `lookup` y `unwind` es el siguiente:

```
Document("\$group", Document()
    .append("_id", "\$planta.nombre_comun")
    .append("total_vendida", Document("\$sum", "\$cantidad"))
    .append("clientes", Document("\$addToSet", "\$cliente.nombre"))
),
Document("\$sort", Document("total_vendida", -1))
```

Prueba y analiza el ejemplo 11

Prueba el código de las consultas del ejemplo y verifica que todas funcionan correctamente.

Práctica 12: Trabaja con tu BD

1. Añade una nueva colección a tu BD.
2. Añade al menú las operaciones CRUD.
3. Programa al menos tres funciones parecidas a las de los ejemplos. Formatea el resultado para que salga `bonito`.
4. Añade al menú de tu aplicación todas las opciones necesarias para poder ejecutar cada función.
5. Recuerda importar y exportar la nueva colección.
6. Crea un fichero `README.md` dentro de la carpeta `main`. Su contenido debe ser un manual de usuario con los siguientes apartados:
    - Descripción general.
    - Requisitos.
    - Base de datos.
    - Cómo ejecutar.
    - Opciones del programa y ejemplos de uso (salida por consola).
    - Notas importantes (si hay algo que destacar).

Entrega 2

Realiza lo siguiente:

1. Asegúrate que tu aplicación exporta correctamente todas las colecciones de tu BD (cada una a un archivo `.json`) y que se guardan en la carpeta `resources` (consulta el apartado `Exportar / Importar la BD con Kotlin` al final de este documento).
2. Entrega en Aules la carpeta `main` de tu proyecto comprimida en formato .zip

IMPORTANTE: El proyecto no debe contener código que no se utilice, ni restos de pruebas de los ejemplos y no debe estar separado por prácticas. Debe ser un proyecto totalmente funcional.

Resumen

Categoría	Comandos clave
Base de datos	`show dbs`, `use`, `db.getName()`, `db.dropDatabase()`
Colecciones	`show collections`, `db.createCollection()`, `db.coleccion.drop()`
Inserción	`db.coleccion.insertOne()`, `db.coleccion.insertMany()`
Consulta	`db.coleccion.find()`, `db.coleccion.findOne()`, `.sort()`, `.limit()`
Actualización	`db.coleccion.updateOne()`, `db.coleccion.updateMany()`, `$set`
Eliminación	`db.coleccion.deleteOne()`, `db.coleccion.deleteMany()`
Índices	`db.coleccion.createIndex()`, `db.coleccion.getIndexes()`, `db.coleccion.dropIndex()`
Estadísticas	`db.stats()`, `db.coleccion.stats()`, `db.version()`

Errores comunes y cómo resolverlos

1) Error: `Error opening socket` / `Unable to connect` - **Causa:** el servidor MongoDB no está en ejecución o la URI es incorrecta. - **Solución:** arrancar el servicio (`sudo systemctl start mongod` en Linux) o comprobar `mongod` en Windows (servicios) y verificar la URI `mongodb://localhost:27017`.

2) `Authentication failed` - **Causa:** autenticación habilitada y credenciales no proporcionadas. - **Solución:** crear un usuario en `admin` con `db.createUser()` o usar una URI con usuario/contraseña: `mongodb://user:pwd@localhost:27017`.

3) `NamespaceNotFound` **/ colección no encontrada** - **Causa:** la colección no existe (no se creó o no tiene documentos). - **Solución:** insertar un documento o crear la colección explícitamente con `db.createCollection("plantas")`.

4) `BSONTypeError` **o problemas de tipo al recuperar datos** - **Causa:** tipos inconsistentes (por ejemplo, altura a veces string, a veces número). - **Solución:** normalizar datos o validar antes de insertar; usar `$convert` en agregaciones o transformar en la app.

5) Problemas con dependencias en Kotlin - **Causa:** dependencia no encontrada o versión incompatible. - **Solución:** comprobar `build.gradle.kts` y usar `mavenCentral()`; actualizar la versión del driver.

6) `No primary found` **en replicación o clúster** - **Causa:** intentando escribir en un conjunto de réplicas sin primario. - **Solución:** comprobar estado del replicaset (`rs.status()`) o arrancar una instancia standalone para prácticas locales.

Exportar / Importar la BD con Kotlin

Desde Kotlin podemos exportar nuestra BD a un archivo .json y también podemos importar un archivo .json a nuestra BD. Para ello hay que añadir la siguiente dependencia en el archivo `build.gradle.kts`.

```
implementation("org.json:json:20231013")
```

A continuación se muestra el código que exporta la BD a un archivo `.json` (actualizado para tomar como parámetros la ruta del `json` y el nombre de la colección).

```
import com.mongodb.client.MongoClients
import org.bson.json.JsonWriterSettings
import java.io.File

fun exportarBD(coleccion: MongoCollection<Document>, rutaJSON: String) {
    val settings = JsonWriterSettings.builder().indent(true).build()
    val file = File(rutaJSON)
    file.printWriter().use { out ->
        out.println("[")
        val cursor = coleccion.find().iterator()
        var first = true
        while (cursor.hasNext()) {
            if (!first) out.println(",")
            val doc = cursor.next()
            out.print(doc.toJson(settings))
            first = false
        }
        out.println("]")
        cursor.close()
```

```
    }

    println("Exportación de ${coleccion.namespace.collectionName} completada")
}
```

A continuación se muestra el código que importa la BD desde un archivo `.json` (actualizado para tomar como parámetros la ruta del `json` y el nombre de la colección).

```
import com.mongodb.client.MongoClients
import org.bson.Document
import org.json.JSONArray
import java.io.File

fun importarBD(rutaJSON: String, coleccion: MongoCollection<Document>) {
    println("Iniciando importación de datos desde JSON...")

    val jsonFile = File(rutaJSON)
    if (!jsonFile.exists()) {
        println("No se encontró el archivo JSON a importar")
        return
    }

    // Leer JSON del archivo
    val jsonText = try {
        jsonFile.readText()
    } catch (e: Exception) {
        println("Error leyendo el archivo JSON: ${e.message}")
        return
    }

    val array = try {
        JSONArray(jsonText)
    } catch (e: Exception) {
        println("Error al parsear JSON: ${e.message}")
        return
    }

    // Convertir JSON a Document y eliminar _id si existe
    val documentos = mutableListOf<Document>()
    for (i in 0 until array.length()) {
        val doc = Document.parse(array.getJSONObject(i).toString())
        doc.remove("_id")  // <-- eliminar _id para que MongoDB genere uno
nuevo
        documentos.add(doc)
    }

    if (documentos.isEmpty()) {
        println("El archivo JSON está vacío")
        return
    }

    val db = cliente.getDatabase(NOM_BD)

    val nombreColeccion =coleccion.namespace.collectionName

    // Borrar colección si existe
```

```
    if (db.listCollectionNames().contains(nombreColeccion)) {
        db.getCollection(nombreColeccion).drop()
        println("Colección '$nombreColeccion' eliminada antes de importar.")
    }

    // Insertar documentos
    try {
        coleccion.insertMany(documentos)
        println("Importación completada: ${documentos.size} documentos de
$nombreColeccion.")
    } catch (e: Exception) {
        println("Error importando documentos: ${e.message}")
    }
}
```

Autoría