CEU Andalucía
Abel Morillo Sánchez

Primero: conceptos claros

¿Qué es una BBDD NoSQL?

Son BBDD que no utilizan el modelo clásico de Entidad-Relación ni el lenguaje SQL. No existen tablas, sino que almacenan documentos que tienen una estructura variable.

VENTAJAS:

- Mucho más eficiente para muchas consultas sobre un volumen de datos muy alto con una estructura similar. Ejemplo: redes sociales.
- Muy fáciles de escalar horizontalmente
- Sencillez
- La estructura de los datos es completamente flexible
- Suelen ser código abierto

INCONVENIENTES:

- No existen restricciones para asegurar la coherencia o integridad de los datos.

¿Qué tipos hay?

- BBDD Clave – Valor. Básicamente almacenan datos binarios identificados por una clave única a través de la que se puede acceder a todo el contenido de forma muy rápida. Ejemplo: Cassandra, Hbase, Redis
- BBDD Documentales. Almacenan la información como si fuera un documento XML o JSON, que tiene un identificador. Además de consultar por esta clave, también se pueden hacer búsquedas por los campos de los documentos. Ejemplo: MongoDB.
- BBDD en grafo. Almacenan la información como nodos de un grafo cuyas relaciones serían las aristas del grafo. Ejemplo: Neo4, InfoGrid
- BBDD orientadas a objetos. Almacenan la información igual que en los lenguajes de programación orientados a objetos (Java, C#, .NET...). Ejemplo: DB4o

¿Qué es MongoDB?

Es una base de datos NoSQL orientada a documentos. Utiliza documentos JSON para crear el esquema de la BBDD. El esquema es completamente flexible, por lo que no tenemos que hacer ningún diseño previo si no queremos. Soporta múltiples lenguajes de programación y es muy utilizada sobre todo en el ámbito industrial.

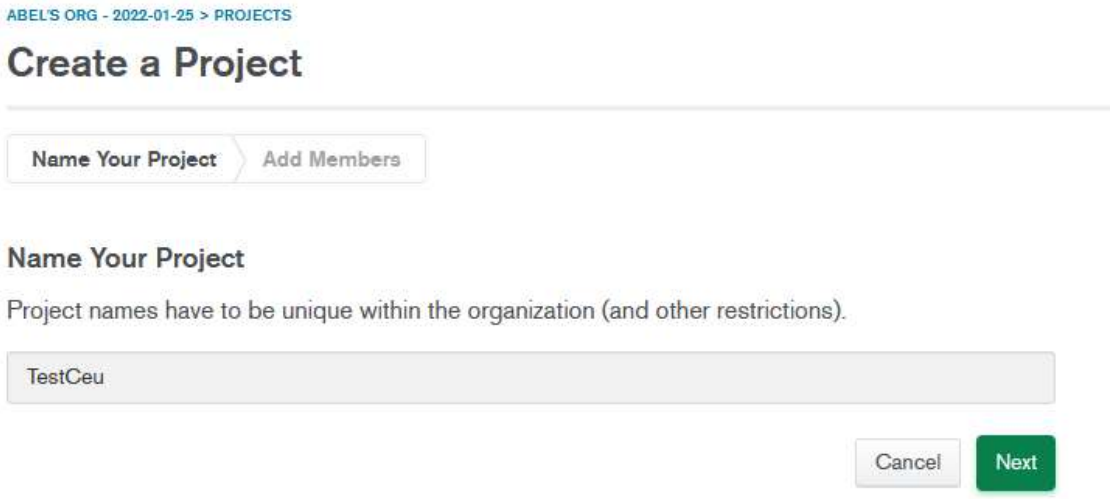
Equivalencia de conceptos

MySQL	MongoDB
Tabla	Collection
Fila o registro	Documento
Columna de una tabla	Campo del documento
Join entre tablas	Documentos enlazados o embebidos

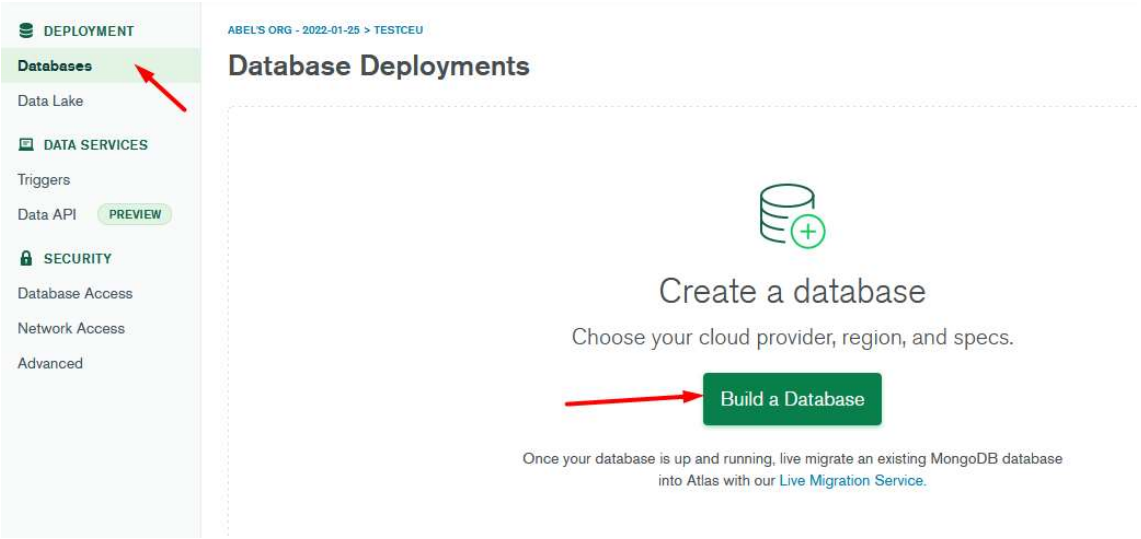
Crear una BBDD MongoDB

MongoDB nos permite crearnos una BBDD en su propio Cloud de forma gratuita para realizar pruebas, por lo que no tendríamos que instalar nada.

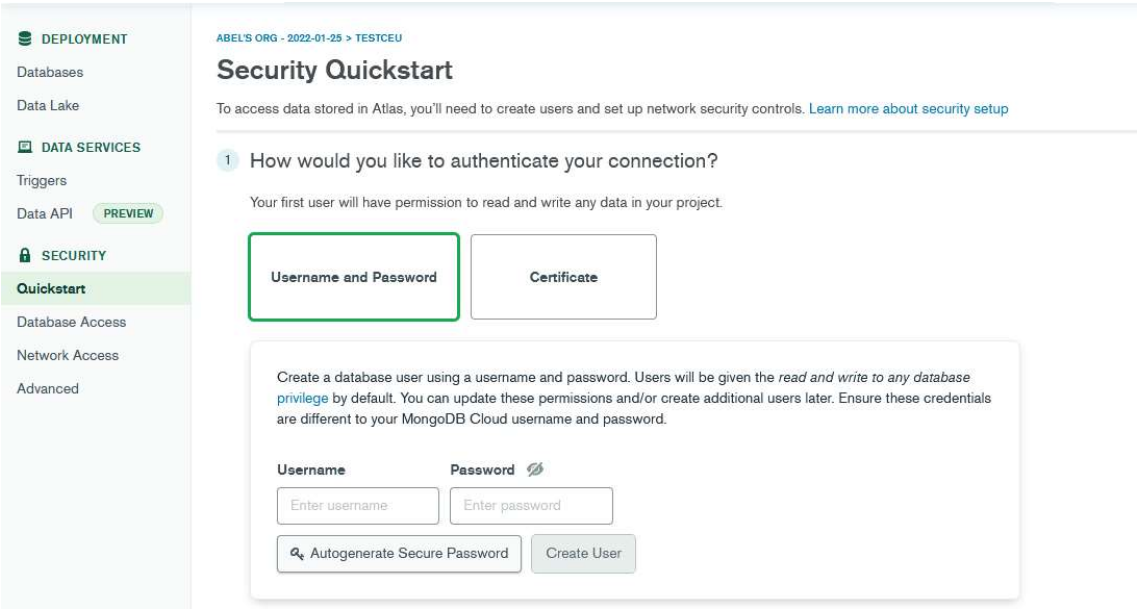
1. Entra en https://www.mongodb.com/es y regístrate
2. Crea un nuevo proyecto:



3. Crea una nueva BBDD. Escoge la opción gratuita y el alojamiento propuesto para el cluster



4. Al crear el cluster, configura usuario y password para la BBDD y la IP desde donde te vas a conectar:



Crear un proyecto Java para trabajar con MongoDB

1. Nos creamos un nuevo proyecto JAVA, lo configuramos como proyecto Maven y añadimos le driver para trabajar con MongoDB:

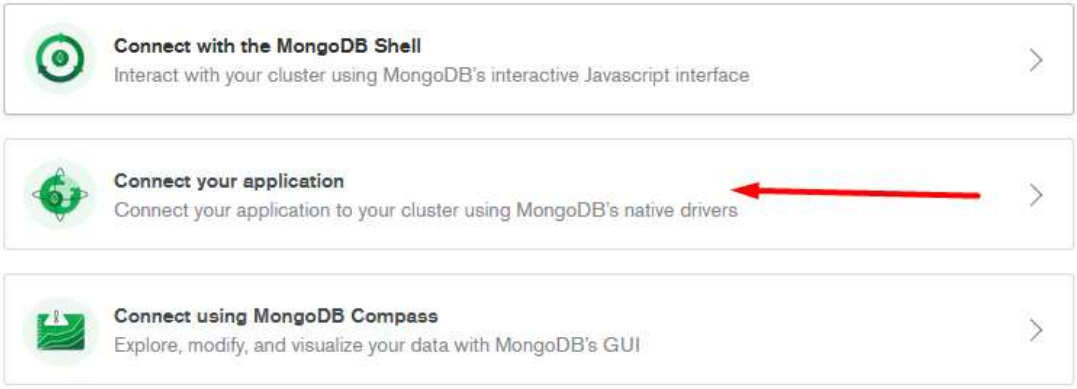
```
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongodb-driver-sync</artifactId>
    <version>4.4.1</version>
</dependency>
```

Hay otro driver llamado “mongo-java-driver”, pero es más antiguo.

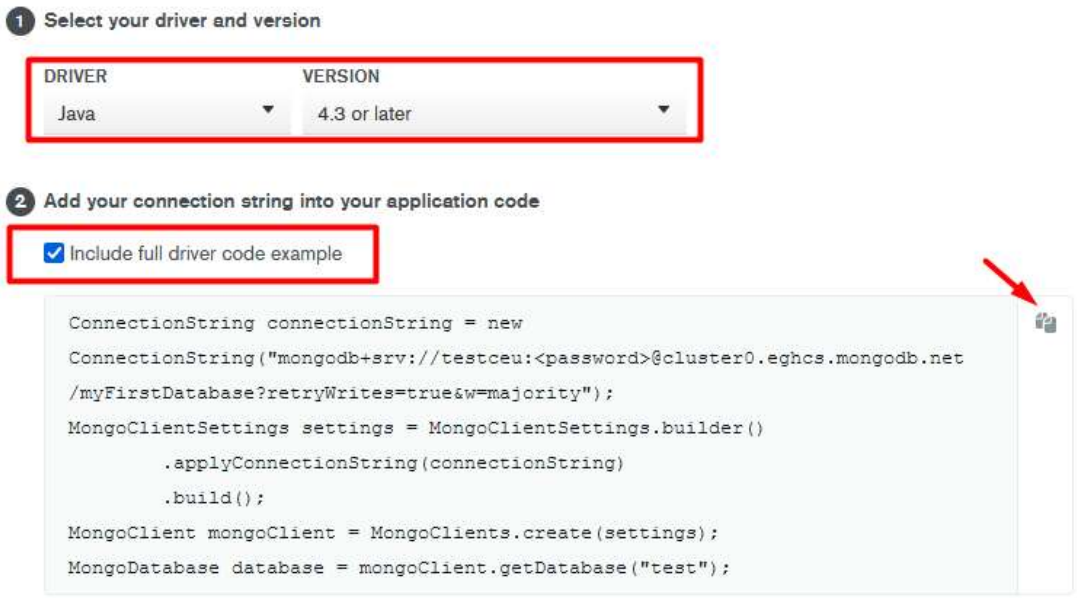
Si queremos que se muestre el log de lo que va haciendo el driver de mongoDB, debemos también añadir esta otra dependencia y configurar completamente el log:

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.35</version>
</dependency>
```

2. Desde MongoDB, seleccionamos la opción “Connect”. Nos aparecerán varias opciones. Escogemos “Connect your applitacion”



Seleccionamos Java versión 4.3 o superior. Y marcamos el check “Include full driver…”



Copiamos el contenido.

3. Nos creamos una clase que nos devuelva la instancia de MongoClient utilizando el código copiado. La clase podría ser algo así:

```
public class MongoUtil {

    private static MongoClient mongoClient;

    public static MongoClient getMongoClient() {
        if (mongoClient == null) {
            initMongoClient();
        }
        return mongoClient;
    }

    private static void initMongoClient() {
        ConnectionString connectionString = new ConnectionString(
                "mongodb+srv://testceu:fuego@cluster0.eghcs.mongodb.net/my
        MongoClientSettings settings = MongoClientSettings.builder().apply
        mongoClient = MongoClients.create(settings);
    }

}
```

Realizar operaciones desde JAVA

Conexión y BBDD

Para cualquier operación que tengamos que hacer, los pasos iniciales siempre serán:

1. Obtener MongoClient. Lo podemos hacer llamando a la clase que hemos creado en el apartado anterior. Eso se conectará a nuestro servidor.
2. Seleccionar la base de datos con la que trabajaremos. Dentro del mismo servidor podemos tener muchas bases de datos. Para seleccionar con la que vamos a trabajar hacemos lo siguiente:

```
MongoClient mongoClient = MongoUtil.getMongoClient();
MongoDatabase db = mongoClient.getDatabase("nombreBaseDatos");
```

Trabajar con colecciones (collections)

Las colecciones son como si fueran tablas. Es donde vamos a guardar documentos. Lo lógico sería tener una colección diferente para cada tipo de documento. Veamos cómo podemos desde JAVA crear/obtener/borrar colecciones:

CREAR

Usamos el método createCollection() e indicamos el nombre. Si ya existe una colección con ese nombre, el método lanzará una excepción:

```
db.createCollection("personas");
```

OBTENER

Usamos el método getCollection(). Nos devolverá un objeto MongoCollection que podemos usar para luego para insertar documentos, buscarlos, etc. La clase MongoCollection requiere que indiquemos el tipo de elementos que contendrá. Por defecto, usaremos la clase Document (de gson). Si la colección no existe, la creará cuando insertemos algún documento. Veamos un ejemplo de uso:

```
MongoCollection<Document> collection =  db.getCollection("nombreColeccion");
```

ELIMINAR

Para eliminar una colección basta con obtenerla y luego llamar al método drop(). Veamos ejemplo:

```
MongoCollection<Document> collection =  db.getCollection("nombreColeccion");
collection.drop();
```

Importante: se llama al método drop() del objeto MongoCollection, no al drop() del objeto MongoDatabase. Si no, borraríamos la base de datos completa.

Insertar documentos

EJEMPLO BÁSICO

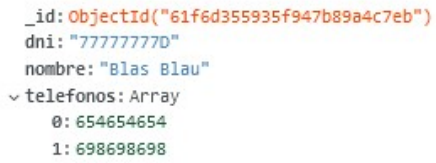
Creamos el documento. Indicamos sus atributos con el método append(). Cuando esté listo, llamamos al método insertOne(). Ejemplo:

```
MongoCollection<Document> collection =  db.getCollection("personas");

Document document = new Document();
document.append("dni", "77777777D");
document.append("nombre", "Blas Blau");
List<Integer> telefonos = new ArrayList<>();
telefonos.add(654654654);
telefonos.add(698698698);
document.append("telefonos", telefonos);

InsertOneResult result = collection.insertOne(document);
System.out.println(result.getInsertedId());
```

Si nos vamos a ver el contenido de mi collection en MongoDB, encontraré esto:



Automáticamente se crea un ID único para cada documento que insertamos. El ID creado lo podemos obtener desde el objeto InsertOneResult que devuelve el insertOne() como se ve en el ejemplo anterior.

DOCUMENTOS ANIDADOS

Podemos insertar documentos que tienen otros documentos anidados. Por ejemplo:

```
MongoCollection<Document> collection =  db.getCollection("personas");

Document document = new Document();
document.append("dni", "77777777D");
document.append("nombre", "Blas Blau");
List<Integer> telefonos = new ArrayList<>();
telefonos.add(654654654);
telefonos.add(698698698);
document.append("telefonos", telefonos);

List<Document> direcciones = new ArrayList<>();
for (int i = 0; i < 3; i++) {
    Document direccion = new Document();
    direccion.append("domicilio", "calle " + i)
            .append("numero", i+10)
            .append("ciudad", "ciudad " + i);
    direcciones.add(direccion);
}
document.append("direcciones", direcciones);

InsertOneResult result = collection.insertOne(document);
System.out.println(result.getInsertedId());
```

El resultado si lo vemos en mongoDB es este:

INSERTAR MUCHOS A LA VEZ

Además del método insertOne() existe el método insertMany() que nos permite insertar muchos documentos a la vez de forma rápida pasando una lista de Document.

En este caso, la lista de IDs generados se retornarán de forma similar en una lista.

GENERAR NUESTRO PROPIO OBJECTID

En lugar de dejar que MongoDB genere automáticamente el ID del documento, podemos generarlo nosotros antes de insertarlo. Lo hacemos instanciando un objeto de la clase ObjectID. Si modificamos el ejemplo anterior, quedaría así:

```
Document document = new Document();
ObjectId objectId = new ObjectId();
System.out.println("Preparando documento con id " + objectId);
document.append("_id", objectId);
document.append("dni", "77777777D");
document.append("nombre", "Blas Blau");
List<Integer> telefonos = new ArrayList<>();
telefonos.add(654654654);
telefonos.add(698698698);
document.append("telefonos", telefonos);
InsertOneResult result = collection.insertOne(document);
System.out.println("Documento insertado con id: " + result.getInsertedId());
```

Los id que se imprimen por consola al principio y al final serían el mismo.

Si intentamos insertar un documento con un ID que ya tiene otro documento, la operación lanzará una MongoWriteException porque la key estará duplicada.

Buscar documentos

Para buscar por documentos utilizaremos el método find(). A este se le pasa siempre un GSON que será un filtro. Este filtro lo podemos construir fácilmente a partir de la clase Filters.

El resultado será siempre un objeto de tipo FindIterable. Sobre este objeto podemos luego realizar múltiples operaciones como ordenar, iterar, etc.

BUSCAR POR ID

Construimos el filtro para localizar el documento con un id determinado. Luego nos quedamos con el primer elemento de los resultados con el método first(). Si el documento no existe, al solicitar el primero, nos devolverá null. Ejemplo:

```
Bson filter = Filters.eq("_id", new ObjectId("61f6db4bb8f2097df2f8d140"));
System.out.println(filter);
FindIterable<Document> result = collection.find(filter);
System.out.println(result.first());
```

OBTENER TODOS LOS DOCUMENTOS

No pasamos ningún filtro en la llamada al find(). Esto nos devolverá todos los documentos de la colección. Para recorrer luego el FindIterable, obtenemos un MongoCursor, y este se podrá recorrer de modo similar a un ResultSet:

```
FindIterable<Document> result = collection.find();
MongoCursor<Document> cursor = result.cursor();
while(cursor.hasNext()) {
    System.out.println(cursor.next().toJson());
}
```

BUSCAR FILTRANDO POR CAMPOS

A partir de aquí, podemos construir las expresiones que queramos para ir aplicando filtros.

>> Ejemplo 1: si quisiera obtener todos los documentos cuyo nombre sea “Laura” o “Blas”, puedo aplicar esta expresión:

```
Bson filter = Filters.or(Filters.eq("nombre", "Laura"), Filters.eq("nombre", "Blas"));
System.out.println(filter);
FindIterable<Document> result = collection.find(filter);
MongoCursor<Document> cursor = result.cursor();
while(cursor.hasNext()) {
    System.out.println(cursor.next().toJson());
}
```

>> Ejemplo 2: todos los documentos que no tienen ningún teléfono:

```
Bson filter = Filters.size("telefonos", 0);
```

>> Ejemplo 3: todos los documentos cuyo nombre contenga "Blas" (debemos usar una expresión regular):

```
Bson filter = Filters.regex("nombre", ".*Blas.*");
```

OTRAS RESTRICCIONES EN LA BÚSQUEDA

Sobre el FindIterable podemos aplicar otro tipo de restricciones u operaciones. Por ejemplo:

- Ordenar los resultados con sort()

```
FindIterable<Document> result = collection.find(filter).sort(Sorts.ascending("dni"));
```

- Limitar los resultados con limit()

```
FindIterable<Document> result = collection.find(filter).limit(3);
```

- Saltar un número de resultados con skip()

```
FindIterable<Document> result = collection.find(filter).skip(3);
```

- Limitar el tiempo máximo que vamos a esperar a que la query termine con maxTime()

```
FindIterable<Document> result = collection.find(filter).maxTime(1, TimeUnit.SECONDS);
```

- Limitar los campos de los documentos devueltos con projection(). Para esto tendremos que crear previamente un BSON con la clase Projections. Veamos ejemplo en el que limito todos los documentos devueltos para que sólo tengan el campo nombre y dni:

```
Bson projectionFields = Projections.fields(
        Projections.include("nombre", "dni"),
        Projections.excludeId());
FindIterable<Document> result = collection.find(filter).projection(projectionFields);
```

CONTAR DOCUMENTOS

Podemos contar los documentos de una colección con dos métodos:

- countDocuments() → Devuelve el número exacto de documentos en un Long. Si tenemos muchos documentos, puede ser un proceso lento. Nos permite además incluir por parámetro un filtro igual que en el método find().

```
MongoCollection<Document> collection =  db.getCollection("personas");
System.out.println(collection.countDocuments());
```

- estimatedDocumentCount() → Devuelve el número estimado de documentos. Es una consulta muy rápida, pero aproximada. En este caso, no se pueden indicar filtros.

```
MongoCollection<Document> collection =  db.getCollection("personas");
System.out.println(collection.estimatedDocumentCount());
```

Actualizar documentos (algunos atributos)

Para actualizar documentos utilizamos el método updateOne() o updateMany() en función de si queremos actualizar sólo un documento (el primero que se adecúe al filtro) o todos.

Los dos métodos reciben dos parámetros:

- Bson filters → Será un filtro igual que el que hemos utilizado en find(). Indicará que documentos queremos actualiza.
- Bson updates → Se trata de un objeto que indica qué queremos actualizar. Lo podemos construir de forma fácil con la clase Updates.

Los dos métodos devuelven un objeto UpdateResult del que podemos obtener el número de documentos actualizados.

>> Ejemplo 1: Actualizar todos los documentos cuyo nombre contenga "Blas" añadiendo el teléfono 621621621 a la lista de teléfonos:

```
MongoCollection<Document> collection =  db.getCollection("personas");
Bson filter = Filters.regex("nombre", ".*Blas.*");
Bson updates = Updates.addToSet("telefonos", 621621621);
UpdateResult result = collection.updateMany(filter, updates);
System.out.println("Documentos acualizados: " + result.getModifiedCount());
```

>> Ejemplo 2: Cambiar el nombre por "Laura"

```
Bson updates = Updates.set("nombre", "Laura");
```

>> Ejemplo 3: Cambiar el nombre por "Laura" y añadir el teléfono 621621621 a la lista:

```
Bson updates = Updates.combine(
                    Updates.set("nombre", "Laura"),
                    Updates.addToSet("telefonos", 621621621));
```

Actualizar documentos (completo)

Para actualizar un documento por completo (sustituir el antiguo por el nuevo) utilizamos el método replaceOne() o replaceMany() en función de si queremos actualizar sólo un documento (el primero que se adecúe al filtro) o todos.

La diferencia con updateOne() y updateMany() es que como segundo parámetro no pasamos otro Bson optenido del Updates, pasamos directamente un Document que será el que sustituya a todos los que cumplan el filtro.

Borrar documentos

Para borrar documentos utilizamos el método deleteOne() o deleteMany() en función de si queremos eliminar sólo un documento (el primero que se adecúe al filtro) o todos.

Los dos métodos reciben un parámetro Bson que será el filtro (igual que el que hemos utilizado en los métodos anteriores). Indicará que documentos queremos borrar. Devuelven un objeto DeleteResult del que podemos obtener el número de documentos eliminados.

Ejemplo:

```
MongoCollection<Document> collection =  db.getCollection("personas");
Bson filter = Filters.eq("nombre", "Laura");
DeleteResult result = collection.deleteMany(filter);
System.out.println("Documentos eliminados: " + result.getDeletedCount());
```

Para borrar todos los documentos, podemos pasar como filtro un documento vacío con new Document().

Trabajar con POJOs en lugar de Document

Lo normal es que los documentos que queramos guardar en nuestra colección muchas veces sean json que representen un objeto de mi modelo. Si es así, podemos trabajar con las clases de nuestro modelo directamente en lugar de con Document. MongoDB se encargará de transformar automáticamente nuestros objetos a Gson y viceversa.

Pero para que esto funcione, tenemos que indicar, al abrir la BBDD, que utilice un codec para trabajar con POJOs. Para no tener que hacerlo de manera continua, lo más cómodo es que nos creemos un método getMongoDB() en nuestra clase MongoUtil que reciba el nombre de la BBDD a la que nos queremos conectar y ya lo haga todo:

```
public static MongoDatabase getMongoDB(String database) {
    CodecRegistry defaultCodecRegistry = MongoClientSettings.getDefaultCodecRegistry();
    CodecProvider codecProvider = PojoCodecProvider.builder().automatic(true).build();
    CodecRegistry pojoCodecRegistry = CodecRegistries.fromProviders(codecProvider);
    CodecRegistry codecRegistry = CodecRegistries.fromRegistries(defaultCodecRegistry, pojoCodecRegistry);
    return getMongoClient().getDatabase(database).withCodecRegistry(codecRegistry);
}
```

Ahora podemos trabajar directamente con objetos de nuestras clases. A continuación, se muestra un ejemplo en el que insertamos una persona y luego consultamos todas las personas de la colección:

```
MongoDatabase db = MongoUtil.getMongoDB("test");

MongoCollection<Persona> collection =  db.getCollection("personas", Persona.class);

Persona p = new Persona("12345678Z", "Blas Blau");
p.getTelefonos().add(654654654);
p.getTelefonos().add(632632632);

InsertOneResult result = collection.insertOne(p);
System.out.println(result.getInsertedId());


FindIterable<Persona> iterable = collection.find();
MongoCursor<Persona> cursor = iterable.cursor();
while (cursor.hasNext()) {
    Persona persona = (Persona) cursor.next();
    System.out.println(persona);
}
```

Eso sí, ahora tenemos que tener cuidado de que los documentos de nuestra colección guarden cierta homogeneidad y que siempre sean una representación de la Clase. Si no, nos encontraríamos con errores.