Acceso a Datos

UT6 – BASES DE DATOS NOSQL

MONGODB DESDE JAVA

1. MongoDB desde Java

Para trabajar en Java con MongoDB necesitamos descarganos el driver, para ello vamos a utilizar la siguiente dependencia de Maven y la añadiremos al archivo pom.xml.

```
<!-- https://mvnrepository.com/artifact/org.mongodb/mongodb-driver-sync -->
    <dependency>
        <groupId>org.mongodb</groupId>
        <artifactId>mongodb-driver-sync</artifactId>
        <version>4.7.0</version>
    </dependency>
```

2. Conexión a la BD

Una vez añadido el driver de Mongodb al proyecto, podemos trabajar ya con las clases del paquete `com.mongodb.client.`

Para conectamos a la base de datos creamos una instancia **`MongoClient,`** por defecto crea una conexión con la base de datos local, y escucha por el `puerto 27017.` Todos los métodos relacionados con operaciones *CRUD (Create, Read, Update and Delete)* en Java se acceden a través de la interfaz **`MongoCollection.`** Las instancias de **`MongoCollection`*`n`*** se pueden obtener a partir de una instancia **`MongoClient`** por medio de una **`MongoDatabase.`**

2. Conexión a la BD

Así pues para **conectarme** a la base de datos *mibasedatos* tendría que importar los siguientes paquetes:

```
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoDatabase;
```

Y el código sería:

```
MongoClient cliente = MongoClients.create();
MongoDatabase db = cliente.getDatabase("mibasedatos");
```

Y para desconectarnos de la base de datos:

```
cliente.close();
```

2. Conexión a la BD

Para conectarnos y desconectarnos a la base de datos deseada vamos a crearnos un método para cada operación. El código sería el siguiente:

```java
package com.iesvjp.mongodb.EjemplosMongoDB;

import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoDatabase;

/**
 * Acceso a MongoDB desde Java
 *
 */
public class App {
    private static MongoClient cliente;

    public static void main(String[] args) {
        MongoDatabase db = connect("pruebas");
        // Una vez establecida la conexión puedo hacer consultas sobre las colecciones de dicha BD
        // showCollection(db, "articulos");
    }

    /**
     *
     * @param database nombre de la base de datos a la que queremos conectarnos
     * @return devuelve un objeto de tipo MongoDatabase
     */
    public static MongoDatabase connect(String database) {
        // Connect to the MongoDB server with internal connection pooling
        cliente = MongoClients.create();
        // Select the database
        MongoDatabase db = cliente.getDatabase(database);
        return db;
    }

    /**
     * Cerramos la conexión con la base de datos
     */
    public static void disconnect() {
        // Disconnect to the MongoDB server
        cliente.close();
    }
}
```

2. Conexión a la BD

Si necesitamos otras opciones (como cambiar el puerto), tenemos toda la información en la documentación oficial:

https://www.mongodb.com/docs/drivers/java/sync/v4.3/fundamentals/connection/mongoclientsettings

3. Visualizar los datos de una colección

Para visualizar los datos de una colección vamos a utilizar la interfaz **`MongoCollection`** (`com.mongodb.client.MongoCollection`): el parámetro de tipo **`Document`** (`org.bson.Document`) es la clase que los clientes utilizan para insertar o modificar los documentos de una colección, y es el tipo predeterminado para devolver búsquedas (***.find)*** y agregados (***.aggregate***). Los datos de la colección se cargan en una lista utilizando el método ***find().into()***

```
/**
 *
 * @param db         parámetro de tipo MongoDatabase
 * @param collection nombre de la colección a consultar
 */
public static void showCollection(MongoDatabase db, String collection) {
    MongoCollection<Document> coleccion = db.getCollection(collection);
    List<Document> consulta = coleccion.find().into(new ArrayList<Document>());
    for (int i = 0; i < consulta.size(); i++) {
        System.out.println(" - " + consulta.get(i).toString());
    }
}
```

3. Visualizar los datos de una colección

La salida por pantalla de la colección artículos sería:

```
- Document{{_id=631a1b292e6a18213916a2b5, codigo=1, denominacion=Portatil Acer, pvp=500, categoria=Informática, uv=10, stock=20}}
- Document{{_id=631a1b292e6a18213916a2b6, codigo=2, denominacion=Pala Pádel, pvp=500, categoria=Deportes, uv=5, stock=30}}
- Document{{_id=631a1b292e6a18213916a2b7, codigo=3, denominacion=Caja Lápices, pvp=6, categoria=Escritorio, uv=10, stock=6}}
- Document{{_id=631a1b292e6a18213916a2b8, codigo=4, denominacion=Marcadores, pvp=10, categoria=Escritorio, uv=20, stock=19}}
- Document{{_id=631a1b292e6a18213916a2b9, codigo=5, denominacion=Memoria 32GB, pvp=120, categoria=Informática, uv=8, stock=10}}
- Document{{_id=631a1b292e6a18213916a2ba, codigo=6, denominacion=Micro Intel, pvp=150, categoria=Informática, uv=4, stock=10}}
- Document{{_id=631a1b292e6a18213916a2bb, codigo=7, denominacion=Bolas Pádel, pvp=5, categoria=Deportes, uv=15, stock=30}}
- Document{{_id=631a1b292e6a18213916a2bc, codigo=8, denominacion=Falda Pádel, pvp=15, categoria=Deportes, uv=10, stock=10}}
```

3. Visualizar los datos de una colección

También podemos recuperar los valores de los campos del documento, utilizando los métodos ***get*** del objeto ***Document,*** reciben como parámetro la clave del dato. Si se sabe el tipo de dato de la clave elegiremos el método correspondiente, y si no utilizamos ***get()*** que devuelve un objeto. Primero cargamos el elemento de la lista en un ***Document.*** Si la clave no existe en el documento visualizará null:

```
public static void showCollectionByFields(MongoDatabase db, String collection) {
    MongoCollection<Document> coleccion = db.getCollection(collection);
    List<Document> consulta = coleccion.find().into(new ArrayList<Document>());
    for (int i = 0; i < consulta.size(); i++) {
        Document amig = consulta.get(i);
        System.out.println(" - " + amig.getString("denominacion") + "-" + amig.get("pvp") + "-"
                + amig.getString("categoria") + "-" + amig.getInteger("uv"));
    }
}
```

3. Visualizar los datos de una colección

Si al ejecutarlo nos muestra el siguiente warning por consola:

```
oct 27, 2022 1:08:36 PM
com.mongodb.diagnostics.logging.Loggers shouldUseSLF4J

ADVERTENCIA: SLF4J not found on the classpath. Logging is
disabled for the 'org.mongodb.driver' component
```

3. Visualizar los datos de una colección

Tenemos que añadir al archivo pom las siguientes dependencias:

```
<dependency>
     <groupId>org.apache.logging.log4j</groupId>
     <artifactId>log4j-api</artifactId>
     <version>2.9.0</version>
</dependency>
<dependency>
     <groupId>ch.qos.logback</groupId>
     <artifactId>logback-classic</artifactId>
     <version>1.2.3</version>
</dependency>
```

3. Visualizar los datos de una colección

Importamos en nuestra clase las siguientes dependencias:

```
import ch.qos.logback.classic.Level;
import ch.qos.logback.classic.Logger;
import ch.qos.logback.classic.LoggerContext;
import org.slf4j.LoggerFactory;
```

Y añadimos las siguientes líneas para evitar los mensajes de warning:

```
LoggerContext loggerContext = (LoggerContext)
LoggerFactory.getILoggerFactory();
Logger rootLogger =
loggerContext.getLogger("org.mongodb.driver");
rootLogger.setLevel(Level.OFF);
```

3. Visualizar los datos de una colección

El método main para los ejemplos que estamos viendo sería:

```
public static void main(String[] args) {

    LoggerContext loggerContext = (LoggerContext) LoggerFactory.getILoggerFactory();
    Logger rootLogger = loggerContext.getLogger("org.mongodb.driver");
    rootLogger.setLevel(Level.OFF);

    MongoDatabase db = connect("pruebas");

    // Una vez establecida la conexión puedo hacer consultas sobre las colecciones
    // de dicha BD
    //showCollection(db, "articulos");
```

4. Visualizar los datos de una colección mapeando BSON a POJO

Lo primero que tenemos que crearnos es nuestra clase POJO a la cual vamos a mapear nuestros BSON. Esta clase debe tener todos los campos posibles que pueden aparecer en nuestro BSON. Y si queremos cambiar el nombre de los atributos tendremos que utilizar la anotación **`@BsonProperty`** tal y como se muestra en el ejemplo.

Por lo tanto, siguiendo nuestro ejemplo, la clase ArticuloPOJO sería:

4. Visualizar los datos de una colección mapeando BSON a POJO

```java
package com.iesvjp.mongodb.EjemplosMongoDB;

import org.bson.codecs.pojo.annotations.BsonProperty;
import org.bson.types.ObjectId;

public class ArticuloPOJO {
    @BsonProperty(value = "_id")
    private ObjectId id;
    private Integer codigo;
    private String denominacion;
    private Integer pvp;
    private String categoria;
    private Integer uv;
    private Integer stock;

    public ObjectId getId() {
        return id;
    }

    public void setId(ObjectId id) {
        this.id = id;
    }

    public Integer getCodigo() {
        return codigo;
    }

    public void setCodigo(Integer codigo) {
        this.codigo = codigo;
    }

    public String getDenominacion() {
        return denominacion;
    }

    public void setDenominacion(String denominacion) {
        this.denominacion = denominacion;
    }

    // Resto de Getters y Setters
```

4. Visualizar los datos de una colección mapeando BSON a POJO

Tal y como hemos visto al principio del tema MongoDB trabaja con documentos BSON. Si queremos trabajar con objetos POJO tendremos que crearnos un *CodecRegistry* a la hora de conectarnos a la BD. Así, podremos trabajar directamente con los POJO para insertar, modificar o leer documentos en la base de datos.

A través del *CodecRegistry* los objetos Java se mapearán automáticamente como documentos a la hora de insertarlos y modificarlos en la colección correspondiente, y también serán mapeados automáticamente a objetos Java desde la colección MongoDB cuando hagamos una lectura.

4. Visualizar los datos de una colección mapeando BSON a POJO

La clase Connection sería:

```
package com.iesvjp.mongodb.EjemplosMongoDB;

import com.mongodb.MongoClientSettings;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoDatabase;
import static org.bson.codecs.configuration.CodecRegistries.fromProviders;
import static org.bson.codecs.configuration.CodecRegistries.fromRegistries;

import org.bson.codecs.configuration.CodecRegistry;
import org.bson.codecs.pojo.PojoCodecProvider;

public class Connection {
    static MongoClient mongoClient;

    public static MongoDatabase connectWithCodecRegistry(String database) {
        mongoClient = MongoClients.create(getClientSettings());
        MongoDatabase db = mongoClient.getDatabase(database);
        return db;
    }

    public static void desconectar() {
        mongoClient.close();
    }

    public static MongoClientSettings getClientSettings() {
        CodecRegistry pojoCodecRegistry = fromProviders(PojoCodecProvider.builder().automatic(true).build());
        CodecRegistry codecRegistry = fromRegistries(MongoClientSettings.getDefaultCodecRegistry(), pojoCodecRegistry);
        MongoClientSettings clientSettings = MongoClientSettings.builder().codecRegistry(codecRegistry).build();
        return clientSettings;
    }
}
```

4. Visualizar los datos de una colección mapeando BSON a POJO

Y ahora el método para visualizar los datos de la colección amigos sería:

```
/**
 *
 * @param db         parámetro de tipo MongoDatabase
 * @param collection nombre de la colección a consultar
 */
public static void showCollectionPOJO(MongoDatabase db, String collection) {
    MongoCollection<AmigoPOJO> amigos = db.getCollection("amigos", AmigoPOJO.class);
    MongoCursor<AmigoPOJO> cursor = amigos.find().iterator();
    while (cursor.hasNext()) {
        AmigoPOJO amigo = cursor.next();
        System.out.println(amigo);
    }
    cursor.close();
}
```

4. Visualizar los datos de una colección mapeando BSON a POJO

Y recuerda que el objeto MongoDataBase que recibe el método se obtiene de llamar al método **`connectWithCodecRegistry()`**

```
MongoDatabase db =
Connection.connectWithCodecRegistry(“pruebas");
```

4. Visualizar los datos de una colección mapeando BSON a POJO

El resultado por consola sería:

```
ArticuloBSON [id=631a1b292e6a18213916a2b5, codigo=1, denominacion=Portatil Acer, pvp=500, categoria=Informática, uv=10, stock=20]
ArticuloBSON [id=631a1b292e6a18213916a2b6, codigo=2, denominacion=Pala Pádel, pvp=500, categoria=Deportes, uv=5, stock=30]
ArticuloBSON [id=631a1b292e6a18213916a2b7, codigo=3, denominacion=Caja Lápices, pvp=6, categoria=Escritorio, uv=10, stock=6]
ArticuloBSON [id=631a1b292e6a18213916a2b8, codigo=4, denominacion=Marcadores, pvp=10, categoria=Escritorio, uv=20, stock=19]
ArticuloBSON [id=631a1b292e6a18213916a2b9, codigo=5, denominacion=Memoria 32GB, pvp=120, categoria=Informática, uv=8, stock=10]
ArticuloBSON [id=631a1b292e6a18213916a2ba, codigo=6, denominacion=Micro Intel, pvp=150, categoria=Informática, uv=4, stock=10]
ArticuloBSON [id=631a1b292e6a18213916a2bb, codigo=7, denominacion=Bolas Pádel, pvp=5, categoria=Deportes, uv=15, stock=30]
ArticuloBSON [id=631a1b292e6a18213916a2bc, codigo=8, denominacion=Falda Pádel, pvp=15, categoria=Deportes, uv=10, stock=10]
```

5. Insertar Documentos - InsertOne

Para insertar documentos, creamos un objeto `Document`, con el método **put** asignamos los pares clave-valor, donde el primer parámetro es la clave, y el segundo el valor. Y con el método `insertOne` se inserta en la colección. El siguiente código añade un amigo a la colección:

5. Insertar Documentos - InsertOne

```java
package com.iesvjp.mongodb.EjemplosMongoDB;

import org.bson.Document;

import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;

public class Insert {

    /**
     *
     * @param db         instancia de la BD
     * @param collection nombre de la colección
     * @param amigo      objeto amigo
     */
    public static void insertarUno(MongoDatabase db, String collection, ArticuloPOJO articulo) {
        MongoCollection<Document> coleccion = db.getCollection(collection);
        Document document = new Document();
        document.put("codigo", articulo.getCodigo());
        document.put("denominacion", articulo.getDenominacion());
        document.put("pvp", articulo.getPvp());
        document.put("categoria", articulo.getCategoria());
        document.put("uv", articulo.getUv());
        document.put("stock", articulo.getStock());
        coleccion.insertOne(document);
    }

}
```

5. Insertar Documentos - InsertOne

En el ejemplo anterior, se ha reutilizado la clase **ArticuloPOJO** añadiendo un constructor por defecto.

Lógicamente, para este apartado nos sirve cualquier clase Java habitual.

5. Insertar Documentos - InsertMany

Se puede insertar en la base de datos una lista de documentos en una colección utilizando el método ***insertMany.*** Además podemos insertar documentos utilizando el método ***append*** de ***Document.***

```
/**
 *
 * @param db         instancia de la BD
 * @param collection nombre de la colección
 * @param lista      ArrayList de objetos articulos
 */
public static void insertarMuchos(MongoDatabase db, String collection, ArrayList<ArticuloPOJO> lista) {
    MongoCollection<Document> coleccion = db.getCollection(collection);
    List<Document> listadocs = new ArrayList<Document>();
    for (ArticuloPOJO articulo : lista) {
        listadocs.add(new Document("codigo", articulo.getCodigo())
                .append("denominacion", articulo.getDenominacion())
                .append("pvp", articulo.getPvp())
                .append("categoria", articulo.getCategoria())
                .append("uv", articulo.getUv())
                .append("stock", articulo.getStock()));
    }
    coleccion.insertMany(listadocs);
    System.out.println("Número de documentos en la colección: " + coleccion.countDocuments());
}
```

6. Consultar documentos

Anteriormente se ha visto cómo cargar los documentos en una lista utilizando el método ***find.into().*** El método ***find()*** devuelve un cursor, una instancia ***FindIterable.*** Podemos utilizar el método ***iterator()*** para recorrer el cursor. En el ejemplo recuperamos todos los documentos de la colección y se visualizan en formato ***Json.***

```
/**
 *
 * @param db         instancia de la BD
 * @param collection nombre de la colección
 */
public static void consultarDocumentos(MongoDatabase db, String collection) {
    MongoCollection<Document> coleccion = db.getCollection(collection);
    MongoCursor<Document> cursor = coleccion.find().iterator();
    while (cursor.hasNext()) {
        Document doc = cursor.next();
        System.out.println(doc.toJson());
    }
    cursor.close();
}
```

6.1. Utilizar filtros en las consultas

El método ***find()*** admite la utilización de filtros. Los filtros son operaciones que usa MongoDB para limitar los resultados que queremos obtener. La clase ***Filters*** proporciona varios métodos estáticos para los diferentes tipos de operaciones que acepta MongoDB. Cada método devuelve un tipo ***BSON***, que luego se puede pasar a cualquier método que espere un filtro de consulta.

Para ello necesitamos importar los siguientes paquetes:

```
import org.bson.conversions.Bson;
import com.mongodb.client.model.Filters;
```

6.1. Utilizar filtros en las consultas

El siguiente ejemplo muestra los productos de la categoría “Informática” con stock entre 20 y 30 incluidas, preguntamos por un intervalo >=20 y <=30:

```java
/**
 *
 * @param db         instancia de la BD
 * @param collection nombre de la colección
 */
public static void consultarProductos_Informatica_20_30(MongoDatabase db, String collection) {
    MongoCollection<Document> coleccion = db.getCollection(collection);
    Bson filter = Filters.and(Filters.eq("categoria", "Informática"), Filters.gte("stock", 20), Filters.lte("stock", 30));
    FindIterable<Document> documents = coleccion.find(filter);
    MongoCursor<Document> cursor = documents.iterator();
    while (cursor.hasNext()) {
        System.out.println(cursor.next());
    }
    cursor.close();
}
```

6.2. Ordenar los resultados

Para ordenar el resultado de una consulta importamos los métodos estáticos de la clase ***Sorts:***

```
import static com.mongodb.client.model.Sorts.*;
```

El siguiente ejemplo muestra los artículos de Informártica ordenados por el campo denominación de forma descendente:

6.2. Ordenar los resultados

```java
/**
 *
 * @param db         instancia de la BD
 * @param collection nombre de la colección
 */
public static void consultarArticulos_Informatica(MongoDatabase db, String collection) {
    MongoCollection<Document> coleccion = db.getCollection(collection);
    Bson filter = Filters.eq("categoria", "Informática");
    MongoCursor<Document> cursor = coleccion.find(filter).sort(ascending("denominacion")).iterator();
    while (cursor.hasNext()) {
        System.out.println(cursor.next());
    }
    cursor.close();
}
```

6.3. Utilizar funciones de agregado

Para utilizar los agregados se necesitan importar los métodos de la clase ***Aggregates***. Cada método devuelve una instancia del tipo BSON, que a su vez se puede pasar al método de agregado de MongoCollection. Y si además queremos utilizar los métodos estáticos de Filters tenemos que importarlos de forma estática:

```
import static com.mongodb.client.model.Filters.*;
import static com.mongodb.client.model.Aggregates.*;
```

6.3. Utilizar funciones de agregado

Para añadir las etapas de agregado podemos utilizar un **Arrays.AsList** de **java.util**. Este ejemplo visualiza los artículos de la categoría Deportes y para ello utiliza la etapa match:

```
/**
 *
 * @param db         instancia de la BD
 * @param collection nombre de la colección
 */
public static void consultaAgregadosMatch(MongoDatabase db, String collection) {
    MongoCollection<Document> coleccion = db.getCollection(collection);
    MongoCursor<Document> cursor = coleccion.aggregate(Arrays.asList(match(eq("categoria", "Deportes")))).iterator();
    while (cursor.hasNext()) {
        System.out.println(cursor.next());
    }
}
```

6.3. Utilizar funciones de agregado

A veces no se necesitan todos los datos contenidos en un documento, se pueden utilizar **proyecciones** para cambiar las salidas. Se necesitan importar los métodos de la clase ***Projection***, estos métodos devuelven un tipo BSON, que podrá ser utilizado en otro método. El import debe ser el siguiente

```
import static com.mongodb.client.model.Projections.*;
```

6.3. Utilizar funciones de agregado

El siguiente ejemplo devuelve la denominación y el precio de los artículos de Deportes, se utiliza el método include para añadir solo denominación y precio y exclude para no mostrar el campo ID:

```
/**
 *
 * @param db         instancia de la BD
 * @param collection nombre de la colección
 */
public static void consultaAgregadosMatchProject(MongoDatabase db, String collection) {
    MongoCollection<Document> coleccion = db.getCollection(collection);
    MongoCursor<Document> cursor = coleccion.aggregate(Arrays.asList(match(eq("categoria", "Deportes")),
            project(fields(include("denominacion", "pvp"), fields(exclude("_id")))))).cursor();
    while (cursor.hasNext()) {
        System.out.println(cursor.next());
    }
    cursor.close();
}
```

6.3. Utilizar funciones de agregado

Si queremos mostrar la media de precio por categoría necesitamos utilizar las funciones de cálculo, para ello tenemos importar los métodos estáticos de la clase Acumulator.

```
import static com.mongodb.client.model.Accumulators.*;
```

```
/**
 *
 * @param db         instancia de la BD
 * @param collection nombre de la colección
 */
public static void consultaAgregadoMediaCurso(MongoDatabase db, String collection) {
    MongoCollection<Document> coleccion = db.getCollection(collection);
    MongoCursor<Document> cursor = coleccion.aggregate(Arrays.asList(group("$categoria", avg("preciomedio", "$pvp"))))
            .cursor();
    while (cursor.hasNext()) {
        System.out.println(cursor.next());
    }
    cursor.close();
}
```

6.3. Utilizar funciones de agregado

Si se desea que la salida de la consulta se almacene en una nueva colección en la base de datos añadimos la etapa ***out.*** En el ejemplo las medias de precio se almacenarán en la colección *preciomedio.*

```
/**
 *
 * @param db         instancia de la BD
 * @param collection nombre de la colección
 */
public static void consultaMediaCursoOut(MongoDatabase db, String collection) {
    MongoCollection<Document> coleccion = db.getCollection(collection);
    coleccion.aggregate(Arrays.asList(group("$categoria", avg("preciomedio", "$pvp")), out("preciomedio"))).toCollection();
    MongoCollection<Document> edadmedia = db.getCollection("preciomedio");
    System.out.println("Número de documentos en la colección: " + edadmedia.countDocuments());
}
```

6.3. Utilizar funciones de agregado

Hemos visto que para añadir las etapas de agregado hemos utilizado únicamente un Arrays.AsList, también podemos utilizar un **Bson** (**org.bson.conversions.Bson**) para cada etapa y luego crearnos la lista, como en el siguiente ejemplo que se calcula el precio medio de los artículos de Escritorio:

```
/**
 *
 * @param db         instancia de la BD
 * @param collection nombre de la colección
 */
public static void getPrecioMedioEscritorio(MongoDatabase db, String collection) {
    MongoCollection<Document> coleccion = db.getCollection(collection);
    Bson match = match(eq("categoria", "Escritorio"));
    Bson group = group("$categoria", avg("preciomedio", "$pvp"));
    Bson sort = sort(descending("categoria"));
    List<Document> results = coleccion.aggregate(Arrays.asList(match, group, sort)).into(new ArrayList<Document>());
    for (int i = 0; i < results.size(); i++) {
        System.out.println(
                "precioMedioEscritorio " + results.get(i).toJson(JsonWriterSettings.builder().indent(true).build()));
    }
}
```

6.3. Utilizar funciones de agregado

Si en la etapa Project necesitamos algún **campo calculado** tendremos que utilizar el método ***computed*** de la factoria ***Projections***, tal y como se muestra en el ejemplo, donde nos creamos el campo importe resultante de multiplicar el precio (pvp) por las unidades vendidas (uv)

```
Bson projectGroup= Aggregates.project( Projections.fields(
     Projections.computed("importe",
     Document.parse("{'$multiply':['$pvp','$uv']}"))
) );
```

6.3. Utilizar funciones de agregado

Tenéis más ejemplos en:

- https://www.mongodb.com/developer/languages/java/java-aggregation-pipeline/?_ga=2.212501074.718979592.1672688954-1442559409.1662224142
- https://www.baeldung.com/java-mongodb-aggregations

7. Actualizar documentos - UpdateOne

Para realizar actualizaciones se necesita importar los métodos de la clase **`Updates y UpdateResult:`**

```
import static com.mongodb.client.model.Updates.*;
import com.mongodb.client.result.UpdateResult;
```

En este ejemplo actualizamos la edad de ***Pepe*** a 20. Para actualizar un único documento se utiliza el método ***updateOne,*** si hay varios con nombre Pepe, actualizará el primero. La actualización devuelve un ***UpdateResult:***

AÑADIR CÓDIGO

7. Actualizar documentos - UpdateOne

```
/**
 *
 * @param db         parámetro de tipo MongoDatabase
 * @param collection nombre de la colección a consultar
 */
public static void actualizarUno(MongoDatabase db, String collection) {
    MongoCollection<Document> coleccion = db.getCollection(collection);
    UpdateResult UpdateResult = coleccion.updateOne(Filters.eq("denominacion", "Portatil Acer"), set("pvp", 501));
    System.out.println("Se han seleccionado: " + UpdateResult.getMatchedCount());
}
```

7. Actualizar documentos - UpdateMany

Si ahora deseamos actualizar varios registros que cumplan una condición utilizamos el método **updateMany**, este ejemplo sube la edad a todos los amigos de 1DAM, la aumenta 1 año. La actualización devuelve un UpdateResult que tiene métodos para decir cuántos se seleccionan y cuántos se actualizan:

```
/**
 *
 * @param db         parámetro de tipo MongoDatabase
 * @param collection nombre de la colección a consultar
 */
public static void actualizarVarios(MongoDatabase db, String collection) {
    MongoCollection<Document> coleccion = db.getCollection(collection);
    UpdateResult UpdateResult = coleccion.updateMany(Filters.eq("categoria", "Informática"), inc("pvp", 1));
    System.out.println("Se han seleccionado: " + UpdateResult.getMatchedCount());
    System.out.println("Se han modificado: " + UpdateResult.getModifiedCount());
}
```

7. Actualizar documentos - UpdateMany

Si se desean actualizar todos los registros de la colección, utilizamos la función ***exists("_id")*** para que devuelvan todos los documentos que tengan ***_id***

```
UpdateResult = colección.updateMany(exists("_id"),inc(“pvp", 1));
```

8. Eliminar documentos

Como en el caso anterior, para borrar un documento de la colección utilizamos el método ***deleteOne,*** y para borrar varios ***deleteMany.*** En el ejemplo se borra el artículo con denominación “Portatil Acer", borrará solo el primero. En el segundo borra todos los documentos de la colección. Devuelven un ***DeleteResult (com.mongodb.client.result.DeleteResult)***

8. Eliminar documentos

```java
/**
 *
 * @param db         parámetro de tipo MongoDatabase
 * @param collection nombre de la colección a consultar
 */
public static void borrarUno(MongoDatabase db, String collection) {
    MongoCollection<Document> coleccion = db.getCollection(collection);
    DeleteResult del = coleccion.deleteOne(Filters.eq("denominacion", "Portatil Acer"));
    System.out.println("Se han borrado: " + del.getDeletedCount());
}

/**
 *
 * @param db         parámetro de tipo MongoDatabase
 * @param collection nombre de la colección a consultar
 */
public static void borrarTodos(MongoDatabase db, String collection) {
    MongoCollection<Document> coleccion = db.getCollection(collection);
    DeleteResult del = coleccion.deleteMany(Filters.exists("_id"));
    System.out.println("Se han borrado: " + del.getDeletedCount());
}
```

Dudas y preguntas