**INDEX**

# TXT y xml:

# JDBC

# SPRING

# Hibernate

# Mongo

## TXT y XML:

Hay dos clases en las que se configura, la clase configuration y el archivo txt o xml del que tira:

* Clase configuration: en el paquete common/config:
* El txt (tambien puede ser un .properties, es lo mismo) o xml del que se basa, con las variables que luego accede:



## Text Files

Son archivos csv que se guardan en la carpeta data en la ruta raíz, los llamas desde el dao.

* **Cargar desde el archivo de texto a memoria**

Antes de nada necesitas una manera de pasar cada línea que veas a un objeto. Se puede hacer de varias maneras, pero lo hacemos con un rowmapper:

@Log4j2  
public class PatientRowMapper {  
 *// Para que lea la fecha 05/05/1990 necesitamos esto:* private static final DateTimeFormatter *DATE\_FORMATTER =* DateTimeFormatter.*ofPattern*("dd/MM/yyyy");

public Patient mapRow(String s) {  
 *// Coges la linea y la divides por el punto y coma* String[] parts = s.split(";");  
 *// Si el tamaño de la cadena es menor a x numero de variables que vamos a usar abajo para crear*

*el objeto se imprime un mensaje de error* if (parts.length < 4) {  
 *log*.error("Invalid input: " + s);  
 return null;  
 }  
 *// Coges los valores de la cadena y los asignas a las variables* try {  
 int id = Integer.*parseInt*(parts[0]);  
 String name = parts[1];  
 LocalDate birthDate = LocalDate.*parse*(parts[2], *DATE\_FORMATTER*);  
 String phone = parts[3];  
 return new Patient(id, name, birthDate, phone);  
 } catch (DateTimeParseException e) {  
 *log*.error("Error parsing date for input: " + s);  
 } catch (NumberFormatException e) {  
 *log*.error("Error parsing number for input: " + s);  
 }  
 return null;  
 }  
}

Ahora, una vez con el rowmapper configurado (y con la instancia del rowmapper inicializada):

*// Metodo que usa el rowmapper*  
*// private final DoctorRowMapper rowMapper;*private Patient mapToPatient(String line) {  
 return rowMapper.mapRow(line);  
}

Una vez todo esto hecho hay dos formas de leer, **para archivos grandes** (BufferedReader):

private List<String> readFile() {  
 List<String> lines = new ArrayList<>();

*// en la configuracion*  
 Path filePath = Paths.*get*(config.getPathPatients());

*// Se lee con bufferedReader en este caso porque es mas*

*eficiente para archivos grandes* try (BufferedReader reader =

Files.*newBufferedReader*(filePath))

{  
 String line;  
 while ((line = reader.readLine()) != null) {  
 lines.add(line);  
 }

} catch (IOException e) {  
 throw new InternalServerErrorException("Error

reading the patient file "+ e);

}  
 return lines;  
}

Y la otra forma, **para archivos pequeños** (usando Files.readAllLines):

@Override  
public List<Doctor> getAll() {  
 List<String> lines = readFile(); *//devuelve una lista de strings que se la paso al rowmapper* return parseDoctors(lines);  
}  
private List<String> readFile() {  
 try { *// Se lee con Files.readAllLines porque es mas sencillo y es un archivo pequeño, si fuera grande se usaria bufferedReader* return Files.*readAllLines*(Paths.*get*(config.getPathDoctors())); *// java.nio.file.Files;* } catch (IOException e) {  
 throw new InternalServerErrorException("Error leyendo el archivo: " + e.getMessage());  
 }  
}  
*//private final DoctorRowMapper rowMapper;*private List<Doctor> parseDoctors(List<String> lines) { *// Mas de lo mismo, se usa el rowmapper para parsear las lineas* return lines.stream()  
 .map(rowMapper::mapRow)  
 .toList();  
}

* **Guardar desde memoria hacia el archivo de texto**

Antes de nada tienes que tener el tostring bien configurado en el objeto que quieras guardar, hazlo con un tostring que no sea el de por defecto por si acaso, aunque lo haga asi yo aqui:

@Override  
public String toString() {  
 return id + ";" + name + ";" +  
 birthDate.format(DateTimeFormatter.*ofPattern*("dd/MM/yyyy"))

+ ";" + phone;  
}

Luego, iteras sobre la lista y vas agregandolos uno a uno, la unica complicacion esá en sacar el ultimo id, pero eso se guarda en una variable cada vez que haces un getAll, llamas al metodo guardarLastId y lo guardas en una variable o algo así.

@Override  
public int save(Doctor d) {  
 int nextIdDoctor= Integer.*parseInt*(config.getNextIdDoctor());  
  
 Path filePath = Paths.*get*(config.getPathDoctors());  
 if (!filePath.toFile().exists()) {  
 throw new InternalServerErrorException("Error saving doctor");  
 } else {  
 try (BufferedWriter bw = Files.*newBufferedWriter*(filePath, *APPEND /\*Para no cargarte lo anterior\*/*)) {  
 bw.write(new Doctor(nextIdDoctor,d.getName(), d.getSpecialty()).toString());  
 bw.newLine();  
 config.setNextIdDoctor(String.*valueOf*(nextIdDoctor+1));  
 return 1;  
 } catch (IOException e) {  
 throw new InternalServerErrorException("Error saving doctor");  
 }  
 }  
}

* **Editar contenido del archivo de texto**

Iteras sobre lo que hay guardado, para que cuando encuentres el mismo objeto que quieres editar, guardes el nuevo en una nueva lista junto con los demas:

@Override  
public void update(Patient updatedPatient) {  
 Path filePath = Paths.*get*(config.getPathPatients());  
 List<String> newLines;  
  
 try (BufferedReader reader = Files.*newBufferedReader*(filePath)) {  
 newLines = reader.lines()  
 .map(line -> {  
 Patient patient = mapToPatient(line);  
 if (patient.getId() == updatedPatient.getId()) {  
 return updatedPatient.toString();  
 } else {  
 return line;  
 }  
 })  
 .toList();  
  
 } catch (IOException e) {  
 throw new InternalServerErrorException("Error reading the patient file " + e);  
 }  
 *//Guardar los cambios, sobreescribiendo lo que ya estaba* try {  
 Files.*write*(filePath, newLines, StandardOpenOption.*TRUNCATE\_EXISTING*);  
 } catch (IOException e) {  
 throw new InternalServerErrorException("Error updating the patient file " + e);  
 }  
}

* **Eliminar contenido del archivo de texto**

Iteras sobre la lista que te acabas de cargar a RAM del getAll, y si el obejto no es el que quieres eliminar, lo vas guardando:

@Override  
public boolean delete(int patientId, boolean confirmation) {  
 List<Patient> patients = getAll();  
 Path filePath = Paths.*get*(config.getPathPatients());  
 try (BufferedWriter writer = Files.*newBufferedWriter*(filePath, StandardOpenOption.*TRUNCATE\_EXISTING*)) {  
 for (Patient patient : patients) {  
 if (patient.getId() != patientId) {  
 writer.write(patient.toString());  
 writer.newLine();  
 }  
 }  
 return true;  
 } catch (IOException e) {  
 throw new InternalServerErrorException("Error deleting the patient " + e);  
 }  
}

## XML FILES

Antes que nada hay que indicar al motor que se encarga de interpretar todo el xml cómo va a ser la estructura del archivo en nuestro modelo de datos. Para ello, necesitamos indicarle exactamente el mismo nombre y la misma estructura que va a haber. Ella te va a dar el modelo de datos, y probablemente pase que los nombres en su xml son distintos a tus variables de tus objetos, que haya elementos en ru modelo de datos que quieras que tu xml ignore o viceversa. Vamos por partes.

Primero, tu lista parte de una raiz en este ejemplo parte de “medRecords”:

*<?*xml version="1.0" encoding="UTF-8" standalone="yes"*?>*<medRecords>  
 <medRecord>  
 <id>2</id>  
 <idPatient>1</idPatient>  
 <doctor>1</doctor>  
 <diagnosis>Hypertension</diagnosis>  
 <date>2010-02-01</date>  
 <medications>  
 <medication>Lisinopril</medication>  
 </medications>  
 </medRecord>  
 <medRecord>  
 <id>3</id>  
 <idPatient>2</idPatient>  
 <doctor>2</doctor>  
 <diagnosis>Asthma</diagnosis>  
 <date>2011-03-01</date>  
 <medications>  
 </medRecord>  
</medRecords>

Entonces, tienes que indicar que todo sale de un “elemento” base llamado medrecords, que dentro de eso hay una lista de “elementos” llamada “medRecord”, y dentro de “medrecord” hay una lista de elementos “medication” llamada “medication”. Se hace así:

@AllArgsConstructor  
@NoArgsConstructor  
@Data  
@XmlRootElement(name="medRecords")  
@XmlAccessorType(XmlAccessType.*FIELD*)  
public class MedRecordsXML {  
 @XmlElement(name = "medRecord")  
 private List<MedRecord> medicalRecords;  
}

El “name” dentro de la etiqueta XmlElement es para que sepa que hay un nombre distinto en el xml que en el modelo de datos, el resto de variables que no tienen esta notacion cogerá el mismo nombre que tienen ya.

@Getter  
@Setter  
@AllArgsConstructor  
@NoArgsConstructor  
@Builder  
@XmlAccessorType(XmlAccessType.*FIELD*)  
public class Medication {  
 @XmlTransient *// Para que no tenga en cuenta esto en el XML.* private int id;  
  
 @XmlElement(name = "medication")  
 private String medicationName;  
  
 @XmlTransient *//Tampoco lo tiene en cuenta en el XML.* private int medRecordId;  
}

@Getter  
@Setter  
@AllArgsConstructor  
@NoArgsConstructor  
@Builder  
@XmlAccessorType(XmlAccessType.*FIELD*)  
public class MedRecord {  
 private int id;  
 private int idPatient;  
 @XmlElement(name = "doctor")  
 private int idDoctor;  
 private String diagnosis;  
 @XmlJavaTypeAdapter(value = LocalDateAdapter.class)  
 private LocalDate date;  
 private List<Medication> medications;  
}

El getAll:

private List<MedRecord> readXML() {  
 File file = new File(config.getPathMedicalRecords());  
 if (!file.exists()) {  
 throw new InternalServerErrorException("Error leyendo el archivo: " + file.getName());  
 }  
 try {  
 *// El contexto en el que le pasas el root element* JAXBContext context = JAXBContext.*newInstance*(MedRecordsXML.class);  
 Unmarshaller unmarshaller = context.createUnmarshaller();  
 *// Desde la raíz, te metes en el elemento que quieres* return ((MedRecordsXML) unmarshaller.unmarshal(file)).getMedicalRecords();  
 } catch (JAXBException e) {  
 throw new InternalServerErrorException("Error while reading the xml, see more details:\n" +  
 e.getMessage());  
 }  
}

El escribir:

private boolean writeXML(List<MedRecord> records) {  
 try {  
 File file = new File(config.getPathMedicalRecords());  
 JAXBContext jaxbContext = JAXBContext.*newInstance*(MedRecordsXML.class);  
 Marshaller marshaller = jaxbContext.createMarshaller();  
 marshaller.setProperty(Marshaller.*JAXB\_FORMATTED\_OUTPUT*, true);  
 *//Pasas el objeto que quieres escribir a root element* MedRecordsXML medRecordsXML = new MedRecordsXML(records);  
 marshaller.marshal(medRecordsXML, file);  
 return true;  
 } catch (JAXBException e) {  
 throw new InternalServerErrorException("Error writing XML"+ e.getMessage());  
 }  
}

El update que llama al escribir una vez que realiza los cambios, igual que antes:

public void update(MedRecord m) {  
 try {  
 List<MedRecord> updatedMedRecords = readXML().stream()  
 .map(medRecord -> {  
 if (medRecord.getId() == m.getId()) {  
 return m;*//si es el mismo, guardo el nuevo* }  
 return medRecord;  
 })  
 .toList();  
 writeXML(updatedMedRecords);  
 } catch (Exception e) {  
 throw new InternalServerErrorException("Error updating the medical record: " + e.getMessage());  
 }  
}

Más de lo mismo con el save:

@Override  
public int save(MedRecord m) {  
 List<MedRecord> medRecords = readXML();  
 int id = Integer.*parseInt*(config.getNextIdMedRecord());  
 m.setId(id);  
 medRecords.add(m);*//lo añades al final con el ultimo id* writeXML(medRecords);  
 if (medRecords.stream().anyMatch(medRecord -> medRecord.getId() == id)) {  
 config.setNextIdMedRecord(String.*valueOf*(id + 1));  
 }else {  
 return 0;  
 }  
 return 1;  
}

El delete:

@Override  
public boolean delete(int id, boolean confirmation) {  
 if (!confirmation) {  
 return false;  
 }  
 List<MedRecord> medRecords = readXML();  
 medRecords.removeIf(m -> m.getId() == id);  
 return writeXML(medRecords);  
}

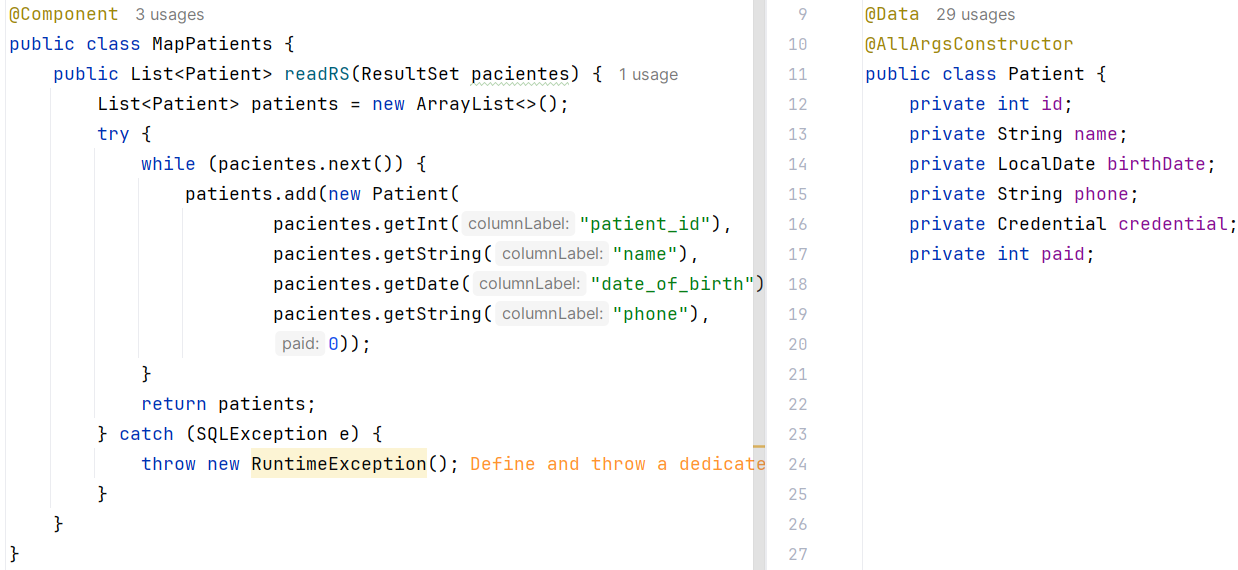
Importante que para que esto vaya bien esté el adapter de localDate bien puesto:

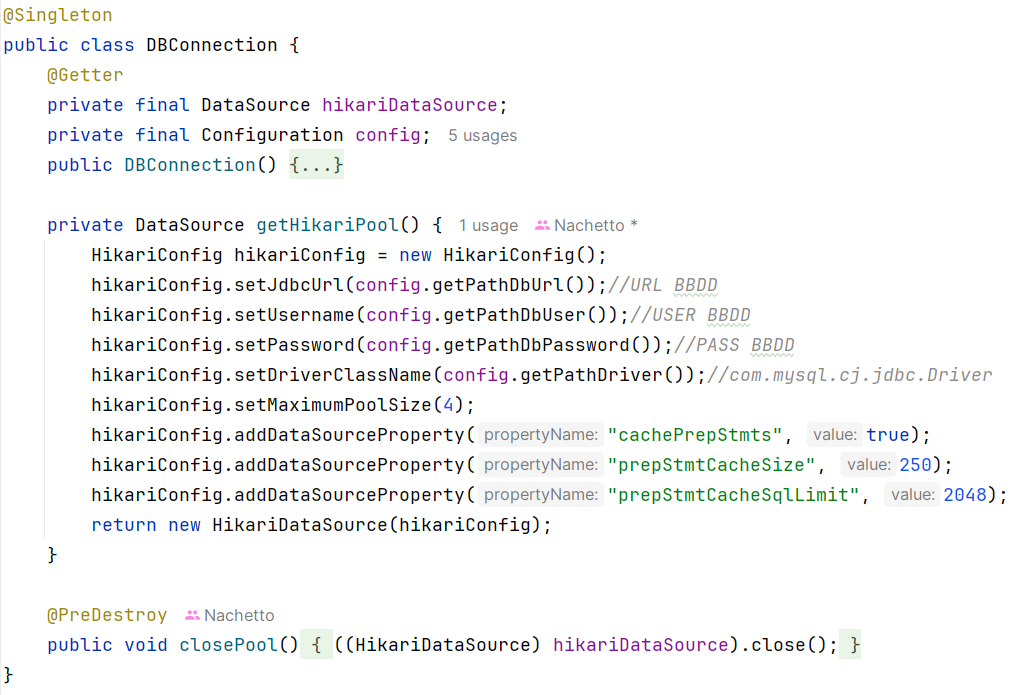
import jakarta.xml.bind.annotation.adapters.XmlAdapter;  
import java.time.LocalDate;  
import java.time.format.DateTimeFormatter;  
import java.time.format.DateTimeParseException;  
  
public class LocalDateAdapter extends XmlAdapter<String, LocalDate> {  
 private static final DateTimeFormatter *DATE\_FORMATTER* = DateTimeFormatter.*ofPattern*("yyyy-MM-dd");  
 public LocalDate unmarshal(String dateString) {  
 try {  
 return LocalDate.*parse*(dateString, *DATE\_FORMATTER*);  
 } catch (DateTimeParseException e) {  
 throw new IllegalArgumentException("Invalid date format" + e.getMessage());  
 }  
 }  
 public String marshal(LocalDate date) {  
 return date.toString();  
 }  
}

## JDBC

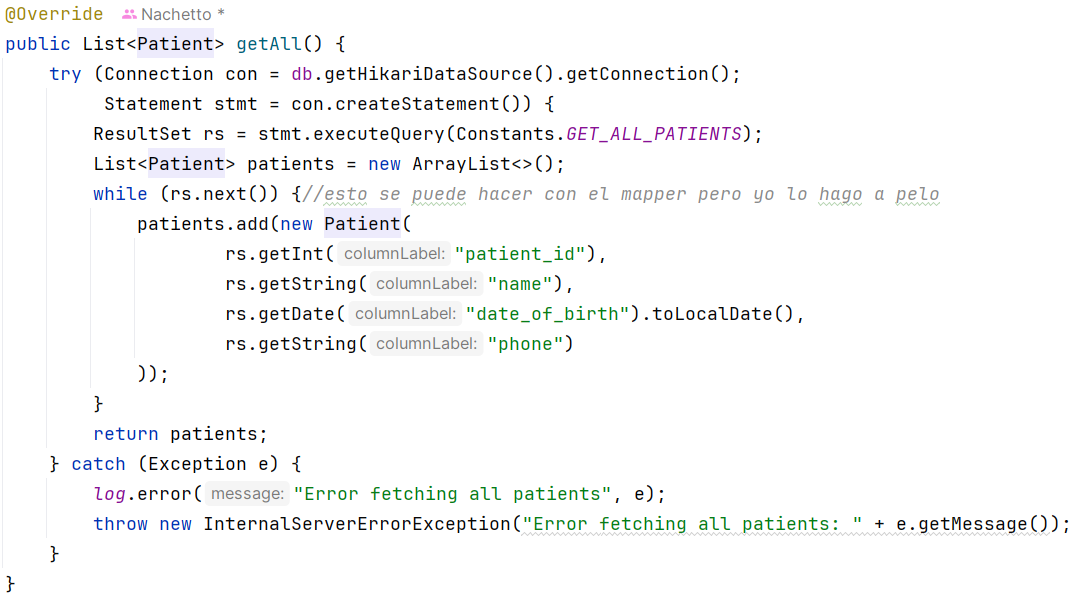
Para que jdbc funcione, tienes que configurar varias cosas, entre otras el ConnectionPool que sea un Singleton y los mapper. El connectionPool básicamente es para que maneje las conexiones automaticamente, los mapper son para pasar lo que recibimos de base de datos a un objeto que pueda entender nuestra aplicacion.

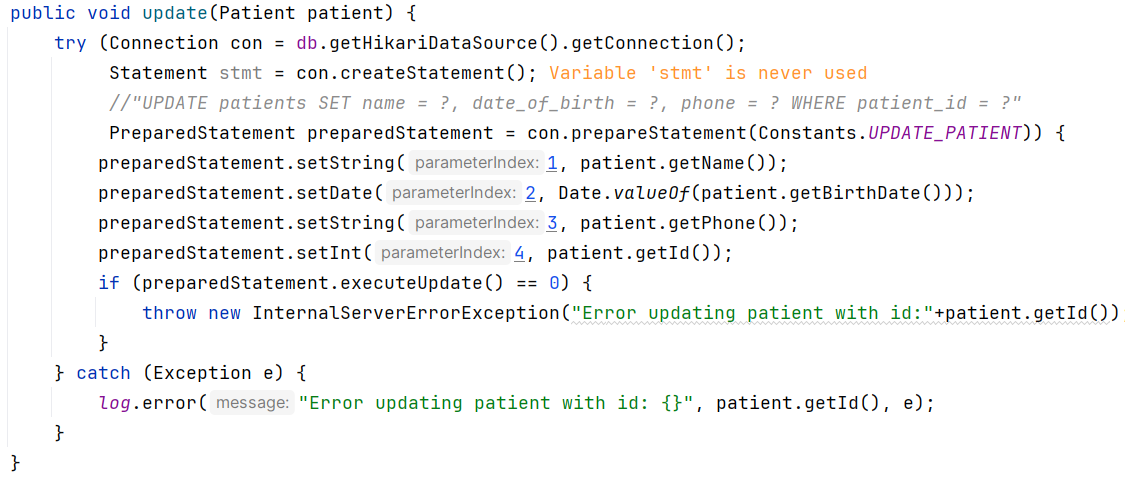
Mapper (objeto de ejemplo al lado) fíjate cómo tienes que poner el nombre de la variable de base de datos:



ConnectionPool muy importante poner el singleton y el predestroy en el closeConnection, para que no se pete:

Una vez con esto vamos al dao, para hacer una llamada tienes que crear el statement, la connection y el resultset que le pasas al mapper:



En ese caso hemos hecho un statement normal, no un preparedStatement, estos segundos son para intercambiar una interrogación en una query por un valor que le especifiques, por ejemplo:

## Hibernate

### Para que funcione:

El Beans, el PersistenceXML, el JpaUtil, la dependencia en el pom

### CRUD:

### Transaccionalidad:

## Mongo

### Para que funcione:

El MongoUtil, la dependencia en el pom

### CRUD:

### Agregations:

Vamos a usar una database sol o porque son consultas get

tenemos varios tipos de busquedas

* van una tras la otra, son como una tuberia, puedes hacer una agregation tipo match y luego debajo una sort, se ordenarían los resultados del match, se llaman stages
* el match que es el común, para hacer un find normal
* los operadores empuiezan con un $ y es como un lenguaje:

$and [variable1:”hola”,variable2:”adios”,variable3: {$gt:10}, …]

* si quieres que te devuelva solo específicos valores, tendrías que hacer un agregation tipo porject, y el id se muestra por defecto:

{

 $match:[nombre:nacho]

 \_id:0, //no lo muestra

nombre:1 //si lo muestra, en este ejemplo acabaría mostrando solo el nombre.

}

* ha nombrado las agregaciones count y sort, simplemente indicas la variable que quieres contar o ordenar (en el caso de sort, variable y 1 si es ascendente, descendente ni idea)
* ha explicado el group, que ni idea, es como el groupBy pero nunca supe hacerlo
* hay que jugar con lo de hacer varios stages para llegar al resultado que desees, por ejemplo, primero las agrupo, luego las ordeno, luego saco la primera
* unwind, que abre el array y crea tantos documentos como elementos del array hayya esto te permite meterte dento del array para poder trabajar con objetos dentro de otros. pero no puedes hacer una stage de project para un objeto dentro de otro, por lo que ese valor lo tienes antes que sacar con otro tipo llamado addfields.
* queda el lookup que es para hacer join, te genera el objeto entero dentro del otro.
* en codigo, el export no puede hacerse porque cre un documento para cada cosa, hay que hacerlo como ella lo tiene el el github

Para los ejercicios.

Primero lo haces en mongo compass, y cuando sepas hacerlo, lo pasas a java pero si no puedes, y haces el export te vale la mitad de la nota.

tienes una clase para todas las agregations, llamada agregationrepository, teniendo un metodo para cada ejercicio, el nombre del metodo tiene que ser el enunciado, solo va a mirar el codigo no el compassttº

1. Aggregation Pipeline

Es una secuencia de etapas (stages) que procesan documentos. Cada etapa filtra, transforma o agrupa los datos, pasando el resultado a la siguiente etapa. Ejemplo de flujo:

$match → $sort → $project → $group

2. Etapas Comunes

a) $match

Filtra documentos, similar a un find.

Ejemplo: Buscar todas las ciudades en Massachusetts (MA) con población mayor a 15,000:

db.zipcodes.aggregate([

  {

    $match: {

      state: "MA",

      pop: { $gt: 15000 }

    }

  }

]);

b) $sort

Ordena los resultados.

* 1: Ascendente
* -1: Descendente

Ejemplo: Ordenar ciudades de MA por población (descendente):

db.zipcodes.aggregate([

  {

    $match: { state: "MA" }

  },

  {

    $sort: { pop: -1 }

  }

]);

c) $project

Selecciona campos específicos.

* 1: Incluir campo
* 0: Excluir campo

Ejemplo: Mostrar solo city y pop, excluyendo \_id:

db.zipcodes.aggregate([

  {

    $project: {

      \_id: 0,

      city: 1,

      pop: 1

    }

  }

]);

d) $count

Cuenta documentos.

Ejemplo: Contar ciudades en MA:

db.zipcodes.aggregate([

  {

    $match: { state: "MA" }

  },

  {

    $count: "total\_ciudades\_MA"

  }

]);

e) $unwind

Descompone un campo que es un array, creando un documento por cada elemento del array. Esto permite trabajar con objetos dentro de arrays.

Ejemplo: Si tienes un documento como este:

{

  "\_id": "01001",

  "city": "AGAWAM",

  "loc": [-72.622739, 42.070206],

  "pop": 15338,

  "state": "MA"

}

Y aplicas $unwind en loc, se generan dos documentos:

{ "\_id": "01001", "city": "AGAWAM", "loc": -72.622739, "pop": 15338, "state": "MA" }

{ "\_id": "01001", "city": "AGAWAM", "loc": 42.070206, "pop": 15338, "state": "MA" }

f) $addFields

Añade nuevos campos o modifica existentes. Es útil para trabajar con objetos anidados, ya que no puedes acceder directamente a ellos en un $project.

Ejemplo: Si tienes un array de objetos:

{

  "\_id": "01001",

  "city": "AGAWAM",

  "details": [

    { "type": "A", "value": 10 },

    { "type": "B", "value": 20 }

  ]

}

Y quieres añadir un campo que sea el doble de value:

db.collection.aggregate([

  {

    $unwind: "$details"

  },

  {

    $addFields: {

      "details.doubleValue": { $multiply: ["$details.value", 2] }

    }

  }

]);

Resultado:

{

  "\_id": "01001",

  "city": "AGAWAM",

  "details": { "type": "A", "value": 10, "doubleValue": 20 }

}

{

  "\_id": "01001",

  "city": "AGAWAM",

  "details": { "type": "B", "value": 20, "doubleValue": 40 }

}

Nota

$unwind es necesario para trabajar con arrays antes de usar $addFields o $project en objetos dentro de ellos.

$project no puede acceder directamente a objetos dentro de arrays sin antes usar $unwind.

3. $group (Agrupación)

Agrupa documentos por un campo y aplica operaciones (suma, promedio, etc.).

* \_id: Campo por el que se agrupa.
* Usa operadores como $sum, $avg, $max.

Ejemplo: Población total por estado:

db.zipcodes.aggregate([

  {

    $group: {

      \_id: "$state",

      total\_poblacion: { $sum: "$pop" }

    }

  }

]);

Resultado:

{ "\_id": "MA", "total\_poblacion": 522901 }

{ "\_id": "NY", "total\_poblacion": 873492 }

4. Combinando Etapas

Ejemplo:  
"Encontrar el estado con mayor población total".

db.zipcodes.aggregate([

  {

    $group: {

      \_id: "$state",

      total\_poblacion: { $sum: "$pop" }

    }

  },

  {

    $sort: { total\_poblacion: -1 }

  },

  {

    $limit: 1

  }

]);

Explicación:

1. Agrupa por estado y suma poblaciones.
2. Ordena de mayor a menor.
3. Limita al primer resultado (el más poblado).

Errores Comunes

* Sintaxis de operadores: Siempre usan $ (ej: $gt, $sum).
* $match incorrecto:  
  ❌ $match: [ { state: "MA" } ] → Correcto: $match: { state: "MA" }.

**EJERCICIOS PARA LA PRÁCTICA**:

**a. Get the oldest patient**

[

  {

    $sort:

      {

        birthDate: 1

      }

  },

  {

    $limit:  1

  }

]

**b. Get the name of the patient who has paid the most**

[

  {

    $unwind:

      {

        path: "$payments"

      }

  },

  {

    $addFields:

      {

        payment: "$payments.amount"

      }

  },

  {

    $group:

      {

        \_id: "$\_id",

        name: {

          $first: "$name"

        },

        totalPayments: {

          $sum: "$payments.amount"

        }

      }

  },

  {

    $sort:

      {

        totalPayments: -1

      }

  },

  {

    $project:

      {

        name: 1,

        totalPayments: 1

      }

  },

  {

    $limit: 1

  }

]