

# UT4 - BBDD Documentales

## RA5 – Gestión de información en bases de datos nativas XML

Desarrollo de aplicaciones que gestionan información almacenada en bases de datos nativas XML mediante el uso de clases específicas.

- Evaluación de las ventajas e inconvenientes de utilizar una base de datos nativa XML.
- Instalación del gestor de base de datos XML.
- Configuración del gestor de base de datos.
- Establecimiento de la conexión con la base de datos.
- Desarrollo de aplicaciones que realizan consultas sobre el contenido de la base de datos.
- Creación y eliminación de colecciones en la base de datos.
- Desarrollo de aplicaciones para añadir, modificar y eliminar documentos XML.

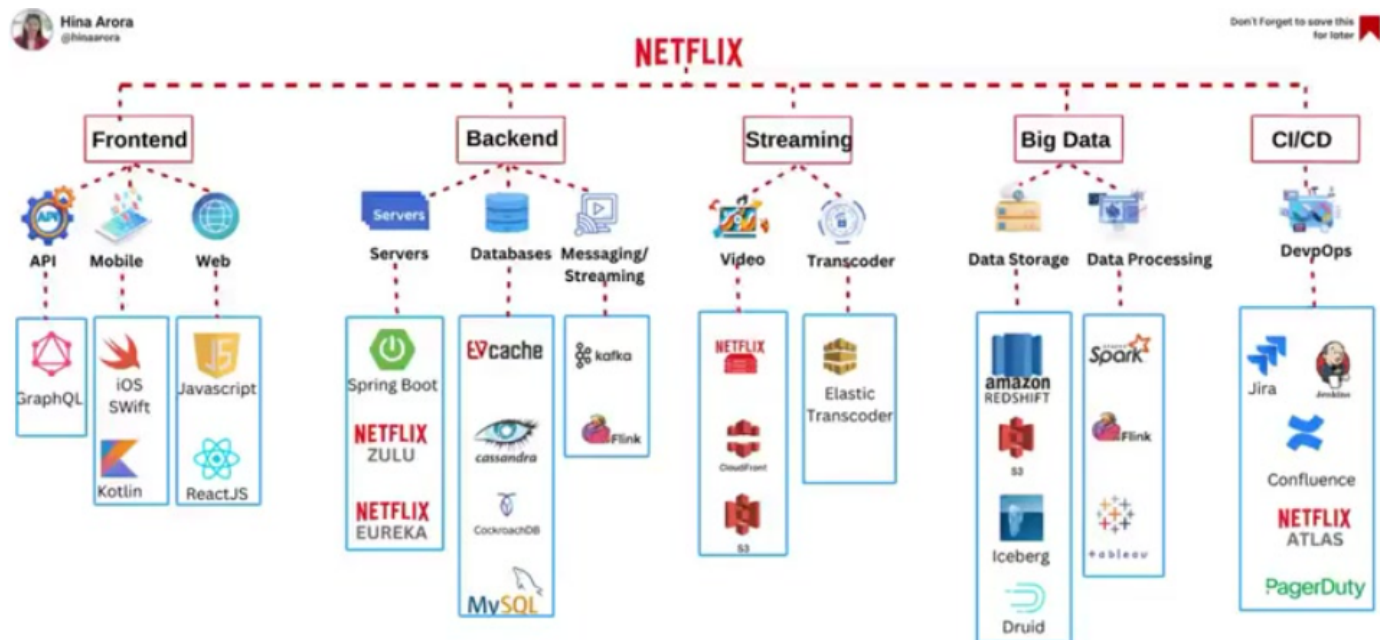
## Modelo híbrido de bases de datos

No todas las bases de datos sirven para todo.

Las aplicaciones reales usan **varios tipos de bases de datos**, cada una para lo que hace mejor. A esto se le llama **modelo híbrido** o **polyglot persistence**.

## Ejemplo real: Netflix

Netflix no usa una única base de datos.



## Otro ejemplo: Aplicación de gestión de estancias en empresa (FCT / Dual)

**Base de datos relacional (MySQL) – Datos críticos** Se utiliza una base de datos relacional para almacenar la información estructurada y crítica del sistema:

- **Alumnos**
- **Empresas**
- **Estancias**

Estos datos requieren **integridad referencial, consistencia y un esquema bien definido**.

**Base de datos NoSQL (MongoDB) – Datos no estructurados** Se utiliza una base de datos orientada a documentos para almacenar información flexible y variable en el tiempo:

- **Mensajes de chat**
- **Incidencias**
- **Eventos de seguimiento**
- **Observaciones del tutor**

Este tipo de información no sigue siempre la misma estructura y puede evolucionar sin necesidad de modificar un esquema fijo.

## Modelos ACID, BASE. Bases de datos relacionales y NoSQL

En una aplicación real pueden convivir **bases de datos relacionales y NoSQL**, cada una para un tipo de dato distinto.

Las BBDD relacionales como **MySQL** siguen el modelo **ACID**:

- **Atomicidad**: una operación se completa entera o se cancela.
- **Consistencia**: los datos siempre cumplen las reglas del sistema, son siempre coherentes.
- **Isolation (Aislamiento)**: las transacciones simultáneas no se interfieren.
- **Durabilidad**: los cambios confirmados no se pierden.

👉 Es la opción adecuada para **datos críticos y estructurados** como alumnos, empresas y estancias.

**MongoDB** es una base de datos documental NoSQL y se asocia al modelo **BASE** en entornos distribuidos:

- **Basically Available**: el sistema prioriza estar disponible sobre la consistencia.
- **Soft State**: el estado puede cambiar mientras se sincronizan los nodos distribuidos.
- **Eventual Consistency**: los datos acaban siendo consistentes con el tiempo, tras la sincronización de todos los nodos.

👉 Es idóneo para **datos flexibles y no estructurados** como chats, incidencias, eventos y observaciones.

**MongoDB es distribuido de forma nativa** porque está diseñado desde el inicio para trabajar en varios nodos mediante **replicación y particionado (sharding)**, lo que facilita la escalabilidad y la alta disponibilidad sin complicar la aplicación.

## Comparativa: Relacional vs NoSQL

Característica	BD Relacional (MySQL)	BD NoSQL (MongoDB)
Modelo de datos	Tablas y filas	Documentos (XML,JSON)
Esquema	Fijo y definido	Flexible y adaptable
Integridad de datos	Muy alta (restricciones, claves)	Menor, se controla desde la aplicación
Transacciones	Completas (ACID)	Soportadas, pero no el foco principal
Relaciones	Claras mediante JOINS	Embebidas o referenciadas
Lecturas	Más complejas con JOINS	Rápidas, datos que no requieren JOINS
Escalabilidad	Vertical principalmente	Horizontal y distribuida nativa
Evolución del modelo	Costosa (migraciones, esquema rígido)	Sencilla (añadir campos)
Ventajas clave	Fiabilidad, coherencia, control	Flexibilidad, rendimiento, escalado
Casos de uso	Datos críticos	Datos con esquema variable y de alto volumen

### Conclusión:

- **Relacional** → cuando prima la coherencia y las relaciones.
- **NoSQL** → cuando prima la flexibilidad y el rendimiento.

## MongoBD

MongoDB es una base de datos **NoSQL orientada a documentos**.

Los datos se organizan en **colecciones** y cada elemento es un **documento** (similar a JSON).

Internamente, MongoDB **no guarda JSON**, sino **BSON (Binary JSON)**.

BSON es un formato **binario** que extiende JSON y permite:

- Tipos de datos reales ( ObjectId , Date , Decimal128 , Binary , etc.).
- Mayor **eficiencia** en almacenamiento y consultas.
- **Indexación** y comparación correcta de datos.

El desarrollador trabaja con documentos tipo JSON, pero MongoDB los almacena y procesa como BSON.

## Instalación Docker

Arranca un contenedor MongoDB con docker-compose.

- Arranca Docker Desktop
- Crea fichero docker-compose.yml

```
services:
  mongo:
    image: mongo
    container_name: mongo-adt
    ports:
      - "27017:27017"
```

- Arrancar contenedor de MongoDB

```
➤ docker compose up -d
# parar con ...
docker compose down
```

## Crear una base de datos con MongoDB Compass

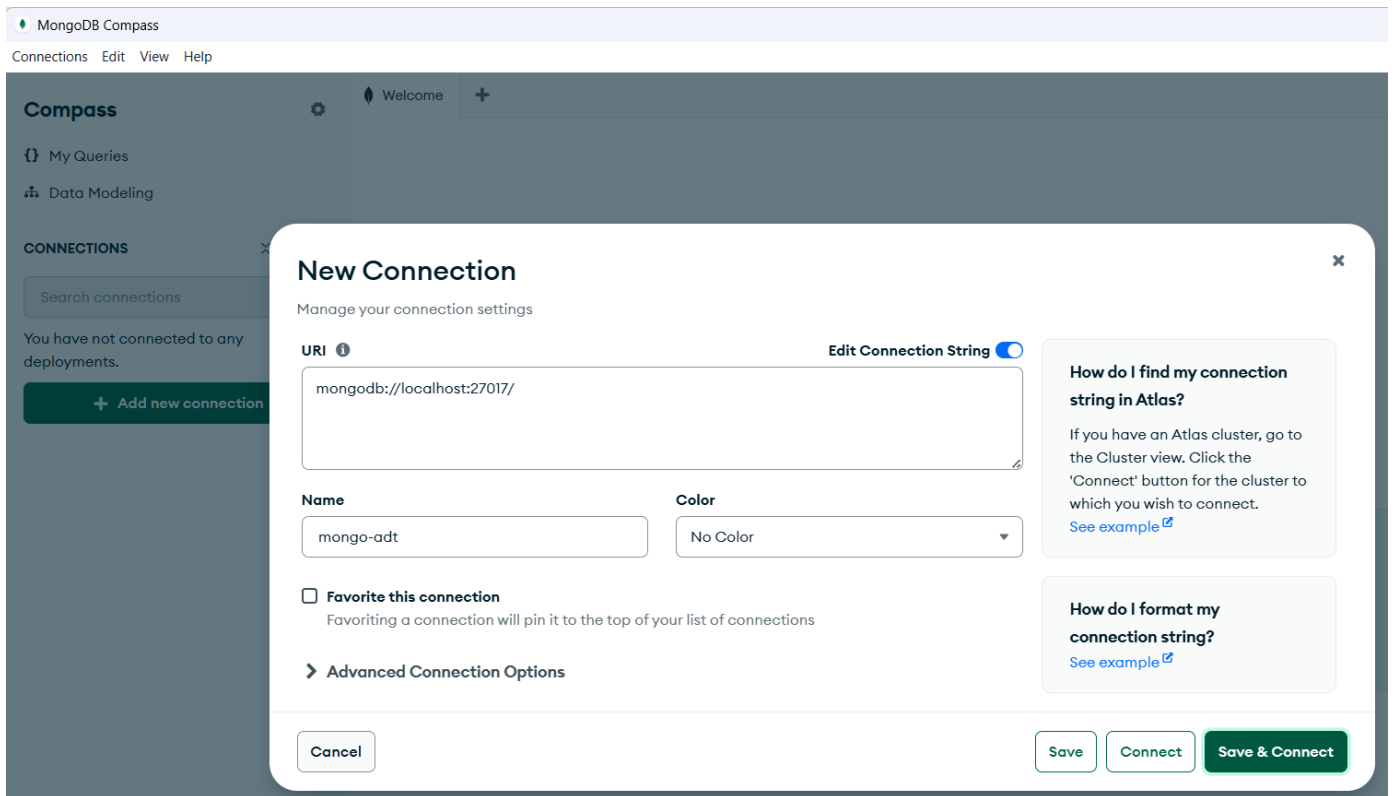
### Instalar MongoDB Compass

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

- Abrir **MongoDB Compass**

### Conectarse al servidor

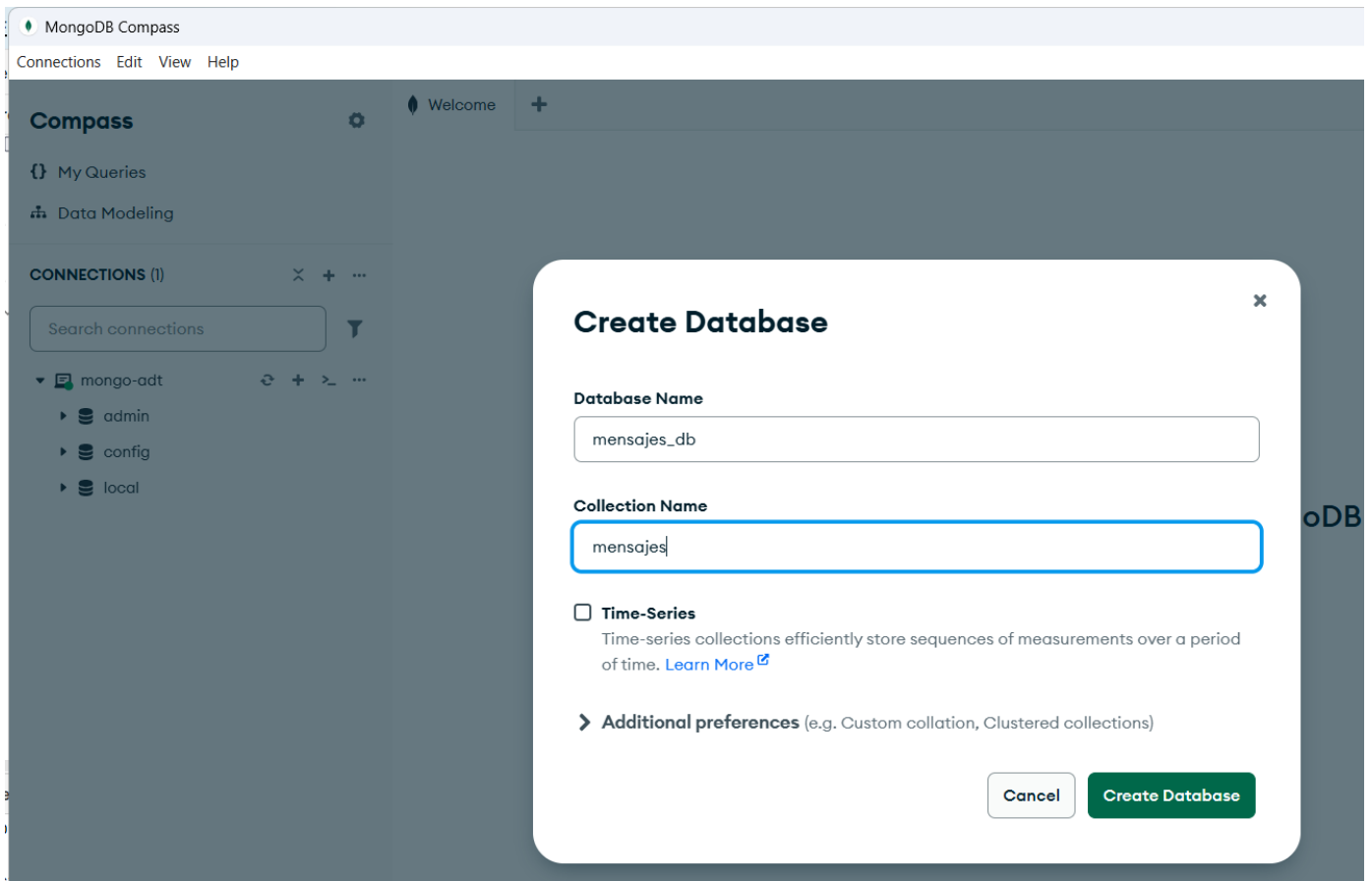
En el campo URI escribir: mongodb://localhost:27017 Pulsar **Connect**



## Crear la base de datos

Vamos a crear una base de datos de ejemplo para almacenar mensajes de chat.

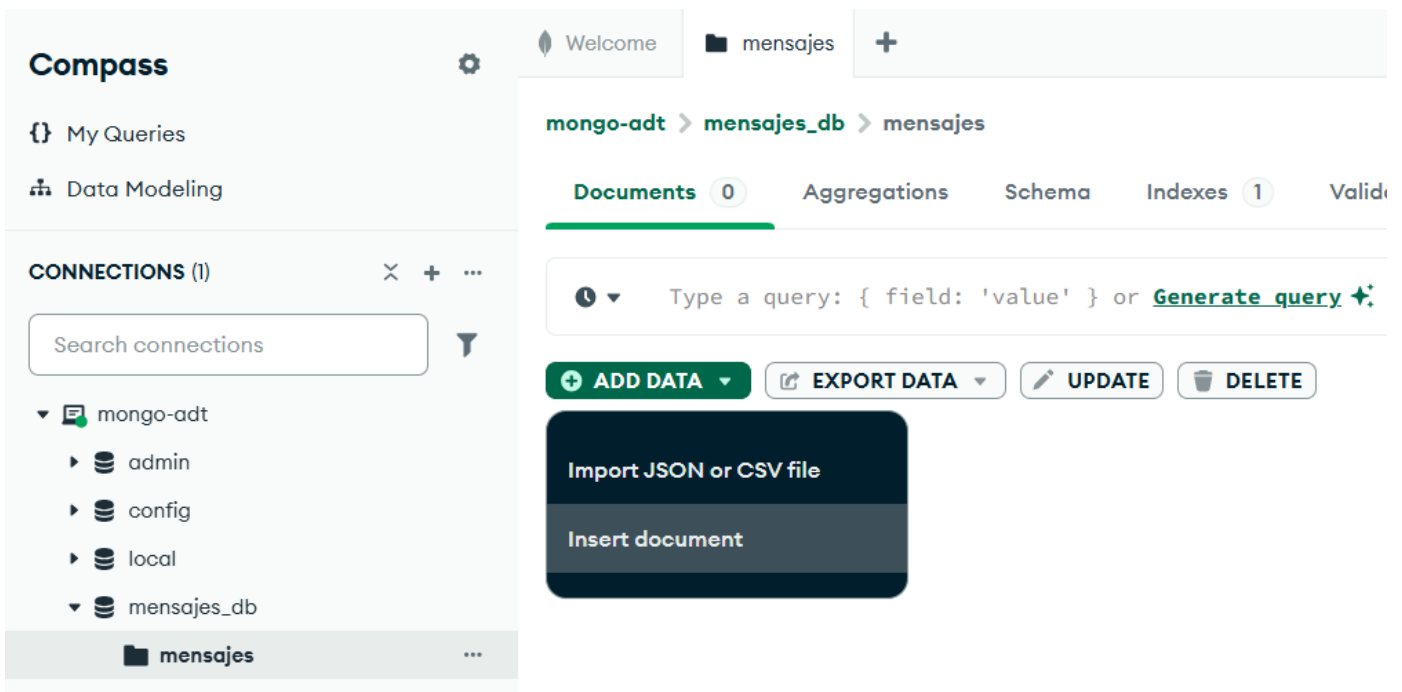
1. Pulsar **Create Database**
2. Rellenar:
  - Database Name: mensajes\_db
  - Collection Name: mensajes
3. Pulsar **Create Database**



👉 En MongoDB la base de datos se crea al crear la primera colección.

## Insertar documentos

1. Entrar en la colección `mensajes`
2. Pulsar **Insert Document**



```
{
  "user": "ana",
  "text": "hola",
  "room": "general"
}
```

Insertar otro:

```
{
  "user": "andres",
  "text": "probando mongo",
  "room": "privado"
}
```

## Cargar datos desde CSV

Sobre la misma coleccion carga los siguientes mensajes. Observa que el esquema es muy flexible, los campos son diferentes y permite la carga de un CSV como el siguiente:

```
user;from;to;datetime;attachments;message
tutor_empresa;Empresa ABC;Juan Pérez;2026-01-05 09:15:00;["foto_incidencia.jpg"];El alumno llega tarde, sin jus
profesor;IES Monte Naranco;Tutor Empresa ABC;2026-01-05 10:30:00;[];Se ha registrado la incidencia, queda pendi
tutor_empresa;Empresa ABC;Juan Pérez;2026-01-06 08:55:00;["justificante.pdf"];Entrega justificante médico, firm
profesor;IES Monte Naranco;Juan Pérez;2026-01-06 12:10:00;[];Revisado el justificante, incidencia cerrada
tutor_empresa;Empresa ABC;Profesor ADT;2026-01-07 14:40:00;["foto_trabajo.jpg","video_proceso.mp4"];Buen rendim
```

## Operaciones desde Mongo Shell

¿Qué es lo que escribimos en la consola mongosh? Se usan **métodos aplicados a una colección**.

- `db` → base de datos actual
- `mensajes` → colección (equivalente a una tabla)
- `find()` , `sort()` , `limit()` → métodos de consulta

Ejemplo conceptual:

```
db.mensajes.find() = buscar documentos en la colección mensajes
```

## Métodos basicos

```
use mensajes_db
```

Acción	Comando en MongoDB Shell
Seleccionar la base de datos	<code>use mensajes_db</code>
Ver un documento	<code>db.mensajes.findOne()</code>
Ver todos los documentos	<code>db.mensajes.find()</code>
Ver solo los primeros documentos	<code>db.mensajes.find().limit(5)</code>
Contar documentos de la colección	<code>db.mensajes.countDocuments()</code>

## Insertar

Insertar 1 documento:

```
db.coleccion.insertOne({ campo: valor })

// Ejemplo:
db.mensajes.insertOne({
  from: "tutor",
  to: "Alumno",
  datetime: "2026-02-05T09:15:00Z",
  attachments: ["foto_incidencia.jpg"],
  message: "El alumno es excelente"
})
```

Insertar varios:

```
db.coleccion.insertMany([ {...}, {...}])
```

## Actualizar

Actualizar un documento:

```
db.coleccion.updateOne(
  { filtro },
  { $set: { campo: nuevoValor } }
)

// Ejemplo:
db.users.updateOne(
  { _id: ObjectId("695ba9e8ffb025418aaf8e46") }, // filtro por ID //
  { $set: { displayName: "Profesor (actualizado)" } }
)
```

Actualizar varios:



```
db.coleccion.updateMany(  
  { filtro },  
  { $set: { campo: nuevoValor } }  
)
```

## Borrar

Borrar un documento:

```
db.coleccion.deleteOne({ filtro })
```

Borrar varios:

```
db.coleccion.deleteMany({ filtro })
```

## Fechas en MongoDB

**Tipo correcto:** `ISODate` ( **BSON Date** )

- En Mongo, una fecha “real” es un **BSON Date** (se guarda como milisegundos desde epoch).
- En shell lo creas así:

```
ISODate("2026-01-05T09:15:00Z")
```

Esto permite ordenar, filtrar por rangos, indexar, etc.

## Filtrados

Operación	Sintaxis pura
Filtrar por un campo	<code>db.&lt;coleccion&gt;.find({ &lt;campo&gt;: &lt;valor&gt; })</code>
Filtrar por otro campo	<code>db.&lt;coleccion&gt;.find({ &lt;campo&gt;: &lt;valor&gt; })</code>
Filtrar por varios campos (AND implícito)	<code>db.&lt;coleccion&gt;.find({ &lt;campo1&gt;: &lt;valor1&gt;, &lt;campo2&gt;: &lt;valor2&gt; })</code>
Filtrar usando OR	<code>db.&lt;coleccion&gt;.find({ \$or: [ { &lt;campo1&gt;: &lt;valor1&gt;, &lt;campo2&gt;: &lt;valor2&gt; }, { &lt;campo1&gt;: &lt;valor1&gt;, &lt;campo2&gt;: &lt;valor2&gt; } ] })</code>

### Ejercicio: Filtrados

Sobre la coleccion de mensajes:

#### 1. Filtrar por un campo (igualdad):

- Muestra todos los mensajes cuyo destinatario ( `to` ) sea **"Juan Pérez"**.

#### 2. Filtrar por otro campo (igualdad):

- Muestra todos los mensajes cuyo origen ( `from` ) sea **"Empresa ABC"**.

#### 3. Filtrar por varios campos (AND implícito):

- Muestra los mensajes cuyo `from` sea **"Empresa ABC"** y cuyo `to` sea **"Juan Pérez"**.

#### 4. Filtrar usando OR:

- Muestra todos los mensajes de la conversación entre **"Empresa ABC"** y **"Juan Pérez"**, tanto si van de empresa→alumno como de alumno→empresa.

## Proyeccion y ordenamiento

Operación	Sintaxis
Proyección	<code>db.&lt;coleccion&gt;.find(&lt;filtro&gt;, { &lt;campo1&gt;: 1, &lt;campo2&gt;: 1, _id: 0 })</code>
Ordenar ascendente	<code>db.&lt;coleccion&gt;.find().sort({ &lt;campo&gt;: 1 })</code>
Ordenar descendente	<code>db.&lt;coleccion&gt;.find().sort({ &lt;campo&gt;: -1 })</code>
Ordenar + proyección	<code>db.&lt;coleccion&gt;.find(&lt;filtro&gt;, &lt;proyeccion&gt;).sort({ &lt;campo&gt;: -1 })</code>
Limitar resultados	<code>db.&lt;coleccion&gt;.find().limit(n)</code>

### Ejercicio: Proyección y ordenamiento

## Enunciado

Usando la colección `mensajes`, escribe las consultas necesarias para:

1. Mostrar solo los campos `datetime`, `from` y `message`, ocultando `_id`.
2. Mostrar los mensajes ordenados por `datetime` de más reciente a más antiguo.
3. Mostrar los 10 mensajes más recientes enseñando únicamente `datetime` y `message`.

## Agregaciones basicas

Operación	Sintaxis
Agregación básica	<code>db.&lt;coleccion&gt;.aggregate([ &lt;etapas&gt; ])</code>
Agrupar	<code>{ \$group: { _id: &lt;campo&gt;, &lt;alias&gt;: { \$sum: 1 } } }</code>
Ordenar resultados	<code>{ \$sort: { &lt;campo&gt;: -1 } }</code>
Limitar resultados	<code>{ \$limit: &lt;n&gt; }</code>

Una **agregación** es una secuencia de etapas (pipeline) que procesa documentos paso a paso.

## Ejercicio: Agregaciones básicas

Usando la colección `mensajes`, escribe las consultas para:

1. Contar cuántos mensajes hay por `user`.
2. Contar cuántos mensajes recibe cada destinatario (`to`).
3. Mostrar los destinatarios ordenados por número de mensajes (de más a menos).
4. Mostrar solo los **3** destinatarios con más mensajes.

## Compass Find. Ver y consultar datos.

Las consultas vistas son aplicables desde el Find de Compass.

Ejemplo: Filtrar por sala

```
{ "room": "general" }
```

mongo-adt > mensajes\_db > mensajes

>\_ Open M...

Documents 2 Aggregations Schema Indexes 1 Validation



{ "room": "general" }

Generate query

Explain

Reset

Find



+ ADD DATA

EXPORT DATA

UPDATE

DELETE



25

1 - 2 of 2



```
_id: ObjectId('695044a6669811f9af958de3')
user: "ana"
text: "hola"
room: "general"
```

```
_id: ObjectId('695044b5669811f9af958de5')
user: "andres"
text: "probando mongo"
room: "general"
```

## Relaciones entre colecciones en MongoDB

En MongoDB, las relaciones se modelan principalmente de 2 formas:

- **Embebidas (embedding)**: un documento contiene a otro (subdocumentos/arrays).
- **Referenciadas (referencing)**: un documento guarda un id que apunta a otro documento.

Además, MongoDB ofrece mecanismos para **resolver** esas referencias:

- `$lookup` (join en agregación, dentro de MongoDB)

# 1) Embedding (relación "dentro del documento")

Consiste guardar los datos relacionados como subdocumentos en el mismo documento.

Ejemplo (1:N embebido):

```
{
  _id: ObjectId("..."),
  cliente: { id: ObjectId("..."), nombre: "Ana" },
  pedidos: [
    { productoId: ObjectId("..."), nombre: "Teclado", precio: 25, cantidad: 2 },
    { productoId: ObjectId("..."), nombre: "Ratón", precio: 15, cantidad: 1 }
  ],
  total: 65
}
```

## Cuándo conviene

- Se consultan juntos casi siempre (lectura frecuente “en bloque”).
- Tamaño controlado (arrays no crecen sin límite).
- Quieres lecturas rápidas sin join.

## Pros

- 1 lectura = obtienes todos los datos (muy rápido).
- Consistencia local fácil (actualización atómica del documento).

## Contras

- Duplicación de datos (p.ej. nombre del producto en muchos pedidos).
- Si el array crece mucho → peor rendimiento y riesgo de límites.

# 2) Referencing (relación por referencia / id)

Idea: guardar un campo que apunta a `_id` de otra colección.

Ejemplo (N:1 típico):

```
// coleccion: pedidos
{
  _id: ObjectId("695f95cda86955baf80b97ae"),
  clienteId: ObjectId("695f95cda86955baf80b97ad"), // Referencia al cliente por id //
  fecha: ISODate("2026-01-08T10:00:00Z")
}

// coleccion: clientes
{
  _id: ObjectId("695f95cda86955baf80b97ad"),
  nombre: "Ana",
  email: "ana@email.com"
}
```

}

## Cuándo conviene

- Relación grande o “compartida” (muchos pedidos → mismo cliente).
- Datos del “padre” cambian y no quieres duplicación.
- Necesitas normalizar (evitar inconsistencias).

## Pros

- Menos duplicación.
- Mejor para entidades reutilizadas y con vida propia.

## Contras

- Para mostrar “pedido + cliente”, necesitas resolver referencia (usando \$lookup desde mongoShell).
- Más lecturas/operaciones si no agregas.

## Ejercicio: Referencing

- Crea una coleccion de usuarios
- inserta 2 usuarios

```
role: "PROFESOR",
username: "profesor",
displayName: "Profesor IES Monte Naranco",
center: "IES Monte Naranco"
```

```
role: "TUTOR_EMPRESA",
username: "tutor_abc",
displayName: "Tutor Empresa ABC",
company: "Empresa ABC"
```

- Inserta un mensaje del profesor al tutor\_empresa utilizando referencias (para ello necesitas conocer los ObjectId insertados)

```
fromUserId: ObjectId("AAA..."),
toUserId: ObjectId("BBB..."),
attachments: [],
message: "Se ha registrado la incidencia, queda pendiente de revisión"
```

# Conectar con MongoDB desde SpringBoot

Dependencia pom.xml

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>

```

application.properties

```

# Conector MongoDB datasource
spring.data.mongodb.uri=mongodb://localhost:27017/mensajes_db

```

## Modelo (Document)

```

@Document(collection = "mensajes")
public class Mensaje {

    @Id
    private String id;

    private String user;
    private String text;
    private String room;
    private Instant datetime;

    // getters, setters y constructores
}

```

## Repository

```

public interface MensajeRepository extends MongoRepository<Mensaje, String> {

    List<Mensaje> findByUser(String user);
    List<Mensaje> findByRoom(String room);
}

```

Para visualizar las sentencias mongo que se ejecutan:

```

# Ver Sentencias Mongo en el log
logging.level.org.springframework.data.mongodb.core.MongoTemplate=DEBUG

```

### Ejercicio: conexion Spring/Mongo

1. Testea desde Junit los metodos CRUD y find

```

class MensajeRepositoryTest {
    @Autowired

```

```
MensajeRepository mensajeRepository;
```

```
@Test  
void create() {  
    // SAVE  
}
```

```
@Test  
void findById() {  
    // SAVE  
    // READ by id  
}
```

```
@Test  
void update() {  
    // SAVE  
    // READ by id  
}
```

```
@Test  
void deleteById() {  
    // CREATE  
    // delete by id  
}
```

```
@Test  
void findByUser() {  
}
```

```
@Test  
void findByRoom() {  
  
}
```

2. Añade al mensaje una coleccion de Attachments.

- i. Para cada adjunto deseamos almacenar: nombre, url y mimeType.
- ii. Crea un mensaje con 2 adjuntos, uno con mimeType image/png y otro application/pdf
- iii. Realiza el test:

```
@Test  
void createWithAttachments() {  
}
```

## Formas de consultar en Spring Data

### A) Derivación por nombre de método

- Spring Data puede **generar consultas** a partir del nombre del método.
- Ventaja: muy rápido para consultas simples.
- Inconveniente: cuando crece la complejidad, el nombre se vuelve largo o insuficiente.

```
// Ejemplos son los vistos anteriormente:  
List<Mensaje> findByUser(String user);  
List<Mensaje> findByRoom(String room);
```

## B) @Query (consulta explícita)

- Permite escribir la consulta Mongo en **JSON** dentro de la anotación.
- Con una sintaxis similar a los filtros realizados desde **mongo shell** con *find()*.
- Ventaja: máximo control y expresividad.
- Inconveniente: requiere conocer la sintaxis de Mongo y es más verboso.
- Parámetros en @Query
  - Se pasan por posición: ?0 , ?1 , ?2 ...
  - Ejemplo: ?0 = primer parámetro del método, ?1 = segundo, etc.

```
// Sintaxis general:  
@Query(  
    value = "{ <consulta_mongo_JSON> }",  
    fields = "{ <proyeccion_JSON> }" // opcional  
)  
TipoRetorno nombreDelMetodo(parametros);
```

### Ejercicio: Consulta @Query con proyeccion y filtrado

Crea un metodo **findMensajesPorMimeType** que devuelva los mensajes con la proyeccion del campo user, y el mimeType pasado por parametro.

- Realiza el test desde Junit y comprueba que devuelve los resultados esperados para el mimeType "application/pdf"



```
@Test
```

```
void findMensajesPorMimeType() {
```

```
    List<Mensaje> mensajes = mensajeRepository.findMensajesPorMimeType ("application/pdf");
```

```
    assertNotNull(mensajes);    mensajes:    size = 1
```

```
}
```

```
}
```

FindMensajesPorMimeType x

Run, Step Through, Breakpoints, Watch, Console, Variables

▼ Evaluate expression (Intro) or add a watch (Ctrl+Mayús+Intro)

MessageRepositoryTest > this = {MensajeRepositoryTest@13994}

MessageRepositoryTest > mensajes = {ArrayList@13993} size = 1

MessageRepositoryTest > 0 = {Mensaje@14007}

MessageRepositoryTest > id = "696138fc7388fa7b4aa89ce0"

MessageRepositoryTest > user = "UserJunit"

MessageRepositoryTest > text = null

MessageRepositoryTest > room = null

MessageRepositoryTest > datetime = null

MessageRepositoryTest > attachments = {ArrayList@14010} size = 0

MessageRepositoryTest > mensajeRepository = {\$Proxy136@13995} "org.springframework.data.mongodb.repository.support."