MEMORIA PROYECTO MONGODB

Almacenamiento y Gestión de la Información

Máster en Ingeniería Informática
Especialidad en Big Data y Cloud Computing

Juan Alberto Domínguez Vázquez

Índice

Introducción

En primer lugar, cabe señalar que, todo el proyecto ha sido realizado en Java con el uso del framework Spring para el desarrollo del backend y con el uso de HTML, CSS y JavaScript para la parte del frontend. El proyecto consiste, por tanto, en una aplicación web alojada en un servidor local Tomcat desplegado por Spring en el puerto 9090 que se conecta a una base de datos MongoDB alojada de forma local también, en el puerto 27017 con dirección IP 127.0.0.1 (localhost). Como filosofía de diseño hemos optado por el uso del patrón MVC (Modelo-Vista-Controlador).

La funcionalidad desarrollada se explicará detalladamente en los siguientes puntos, pero consiste en operaciones CRUD para cada una de las colecciones del dataset YELP propuesto en el enunciado de la práctica, superando, por tanto, la cantidad de 8 operaciones pedidas en dicho enunciado, así como las acciones obligatorias que detallamos a continuación:

- Creación de índices sobre los campos text de las colecciones Tip y Review, ambos de tipo texto, sobre los campos stars, business_id y categories de la colección Business y sobre el campo name de la colección User.
- Dos consultas sobre sobre el campo text de la colección Review, la primera es buscar palabras clave en el campo 'text' con paginación y la segunda es buscar con múltiples palabras clave ordenadas por relevancia con paginación, cada una con 3 parámetros.
- Dos consultas usando el framework de agregación de Mongo, que son getTopRatedBusinessesByCategory y getBusinessesWithMostReviewsByCategory ambas con un parámetro cada una.
- Dos consultas que impliquen 2 o más colecciones relacionadas por "referencia" que son getInvoicesForBusiness y getReviewsByUserAndDate ambas con dos parámetros cada una.

En cuanto a los problemas detectados en el enunciado con respecto a las colecciones se han arreglado lo siguientes:

El campo "categories" de la colección "business" debería ser un array. Sin embargo, viene como una cadena de elementos separados por coma:

```
db.business.updateMany(
  { categories: { $exists: true, $type: "string" } },
  [
    {
      $set: {
        categories: { $split: ["$categories", ", "] }
      }
    }
  ]
);
```

El campo "friends" de la colección "user" debería ser un array. Sin embargo, viene como una cadena de elementos separados por coma:

```
db.user.updateMany(
  { "elite": { $type: "string" } },
  [
    {
      $set: {
        elite: { $split: [{ $trim: { input: "$elite" } }, ","] }
      }
    }
  ]
);
```

El campo "elite" de la colección "user" debería ser un array. Sin embargo, viene como una cadena de elementos separados por coma:

```
db.user.updateMany(
  { "friends": { $type: "string" } },
  [
    {
      $set: {
        friends: { $split: [{ $trim: { input: "$friends" } }, ","] }
      }
    }
  ]
);
```

Dentro del campo "attributes" de la colección "business", hay campos que deberían ser documentos. Sin embargo, está entre comillas y, por tanto, lo trata como una cadena.

Para solucionar este problema se ha creado el script llamado scriptProyecto, cuya estructura se explica a continuación y que debe ser lanzado desde mongosh, conectándonos a la base de datos yelp previamente, con el comando: load("rutaAlArchivo/scriptProyecto.js");

```
// Conectar a la base de datos
const db = connect('mongodb://localhost:27017/yelp');

// Función para convertir una cadena a un objeto JSON válido
function convertToDocumentEmbedded(str) {
  // Mostrar la cadena para depurar
  print(`Intentando convertir la cadena: ${str}`);

  // Reemplazar comillas simples por dobles para hacerla compatible con JSON
  let jsonStr = str.replace(/'/g, '"');

  // Eliminar el prefijo 'u' en las claves
  jsonStr = jsonStr.replace(/u"([^"]+)"/g, '"$1"');

  // Reemplazar los valores True/False por true/false de JSON
  jsonStr = jsonStr.replace(/True/g, 'true').replace(/False/g, 'false');

  // Reemplazar el valor None por null de JSON
  jsonStr = jsonStr.replace(/None/g, 'null');

  // Intentar parsear el JSON
  try {
    const jsonObject = JSON.parse(jsonStr);
    return jsonObject;
  } catch (e) {
    print(`Error al convertir la cadena a objeto JSON: ${str}`);
    return null; // Devolver null si no se puede convertir
  }
}

// Encontrar documentos con los campos incorrectos
const cursor = db.business.find({
  $or: [
    { "attributes.BusinessParking": { $type: "string" } },
    { "attributes.GoodForMeal": { $type: "string" } },
    { "attributes.Ambience": { $type: "string" } },
    { "attributes.Music": { $type: "string" } }
  ]
});
```

```
// Iterar sobre los documentos y corregir los valores
cursor.forEach(doc => {

  const updateFields = {};
  let updated = false;

  // Convertir BusinessParking de cadena a objeto embebido si es necesario
  if (typeof doc.attributes.BusinessParking === 'string') {
    const converted = convertToDocumentEmbedded(doc.attributes.BusinessParking);
    if (converted) {
      updateFields["attributes.BusinessParking"] = converted;
      updated = true;
    }
  }

  // Convertir GoodForMeal de cadena a objeto embebido si es necesario
  if (typeof doc.attributes.GoodForMeal === 'string') {
    const converted = convertToDocumentEmbedded(doc.attributes.GoodForMeal);
    if (converted) {
      updateFields["attributes.GoodForMeal"] = converted;
      updated = true;
    }
  }

  // Convertir Ambience de cadena a objeto embebido si es necesario
  if (typeof doc.attributes.Ambience === 'string') {
    const converted = convertToDocumentEmbedded(doc.attributes.Ambience);
    if (converted) {
      updateFields["attributes.Ambience"] = converted;
      updated = true;
    }
  }

  // Convertir Music de cadena a objeto embebido si es necesario
  if (typeof doc.attributes.Music === 'string') {
    const converted = convertToDocumentEmbedded(doc.attributes.Music);
    if (converted) {
      updateFields["attributes.Music"] = converted;
      updated = true;
    }
  }
```

```
  // Realizar la actualización si es necesario
  if (updated) {
    db.business.updateOne(
      { _id: doc._id },
      { $set: updateFields }
    );
  } else {
    print(`No se realizó actualización para el documento con _id: ${doc._id}`);
  }
});
```

Diseño y código de la aplicación

Como se ha explicado en el apartado anterior, se optado por seguir el patrón MVC integrado con Spring, por tanto, el proyecto posee varios niveles de paquetes:

- Entidades
- Repositorios
- Servicios
- Controladores
- Vistas

Por otra parte, también tenemos algunas clases de configuraciones necesarias para el correcto funcionamiento de MongoDB y su completa integración con Spring, así como clases para validación de errores, de esquemas y convertidores de tipos. Como, por ejemplo, un convertidor de String al tipo Java LocalDateTime, o una clase para la validación de las entidades según sus anotaciones.

En cuanto a las entidades, tenemos 5, una por cada colección siendo estas, Review, Tip, User, Business y Checkin. Todas ellas han sido anotadas para su integración con Mongo como vemos a continuación:

Entidad Business:

```java
@Document(collection = "business")
public class Business
{
    @Id
    private String id;

    @NotBlank(message = "La id del negocio es obligatorio.")
    @Field(name = "business_id")
    private String businessId;

    @NotBlank(message = "El nombre del negocio es obligatorio.")
    private String name;

    @NotBlank(message = "La direccion es obligatoria  y debe ser una cadena.")
    private String address;

    @NotBlank(message = "La ciudad es obligatoria y debe ser una cadena.")
    private String city;

    @NotBlank(message = "El estado es obligatorio y debe ser una cadena.")
    private String state;

    @NotBlank(message = "El codigo postal es obligatorio y debe ser una cadena.")
    @Field("postal_code")
    private String postalCode;

    @Min(value = -90, message = "La latitud debe estar entre -90 y 90.")
    @Max(value = 90, message = "La latitud debe estar entre -90 y 90.")
    private double latitude;
```

```
@Min(value = -180, message = "La longitud debe estar entre -180 y 180.")
@Max(value = 180, message = "La longitud debe estar entre -180 y 180.")
private double longitude;

@Min(value = 0, message = "Las estrellas deben estar entre 0 y 5.")
@Max(value = 5, message = "Las estrellas deben estar entre 0 y 5.")
private double stars;

@Min(value = 0, message = "El numero de reseñas debe ser un entero no negativo.")
@Field("review_count")
private int reviewCount;

@NotNull(message = "El estado de apertura es obligatorio.")
@Min(value = 0, message = "El valor de is_open debe ser 0 o 1.")
@Max(value = 1, message = "El valor de is_open debe ser 0 o 1.")
@Field("is_open")
private int isOpen;

private Map<String, Object> attributes;

private List<String> categories;

private Map<String, String> hours;
```

Entidad Checkin:

```
@Document(collection = "checkin")
public class Checkin
{

    @Id
    private String id;

    @NotBlank(message = "El business_id es obligatorio.")
    @Field("business_id")
    private String businessId;

    @NotNull(message = "La fecha no puede ser nula.")
    @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    private List<LocalDateTime> date;
```

Entidad Review:

```java
@Document(collection = "review")
public class Review
{

    @Id
    private String id;

    @Field("review_id")
    @NotBlank(message = "El review_id es obligatorio.")
    private String reviewId;

    @Field("user_id")
    @NotBlank(message = "El user_id es obligatorio.")
    private String userId;

    @Field("business_id")
    @NotBlank(message = "El business_id es obligatorio.")
    private String businessId;

    @Min(value = 0, message = "Las estrellas deben ser al menos 1.")
    @Max(value = 5, message = "Las estrellas no pueden ser mayores a 5.")
    private float stars;

    @Min(value = 0, message = "El valor de useful debe ser 0 o mayor.")
    private int useful;

    @Min(value = 0, message = "El valor de funny debe ser 0 o mayor.")
    private int funny;
```

```java
@Min(value = 0, message = "El valor de cool debe ser 0 o mayor.")
private int cool;

@NotBlank(message = "El texto de la reseña no puede estar vacío.")
private String text;

@NotNull(message = "La fecha es obligatoria.")
@DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
private LocalDateTime date;
```

Entidad Tip:

```java
@Document(collection = "tip")
public class Tip
{

    @Id
    private String id;

    @Field("user_id")
    @NotBlank
    private String userId;

    @Field("business_id")
    @NotBlank
    private String businessId;

    @Field("text")
    @NotBlank
    private String text;

    @Field("date")
    @NotNull
    private LocalDateTime date;

    @Field("compliment_count")
    @Min(value = 0, message = "Los complimentCount han de ser minimo 0.")
    private int complimentCount;
```

Entidad User:

```java
@Document(collection = "user")
public class User
{

    @Id
    private String id;

    @Field("user_id")
    @NotBlank
    private String userId;

    @Field("name")
    @NotBlank
    private String name;

    @Field("review_count")
    @Min(value = 0, message = "El contador de visitas ha de ser minimo 0.")
    private int reviewCount;

    @Field("yelping_since")
    @NotBlank
    private String yelpingSince;

    @Field("useful")
    @Min(value = 0, message = "Useful ha de ser minimo 0.")
    private int useful;
```

```java
@Field("funny")
@Min(value = 0, message = "Funny ha de ser minimo 0.")
private int funny;

@Field("cool")
@Min(value = 0, message = "Cool ha de ser minimo 0.")
private int cool;

private List<String> elite;
private List<String> friends;

@Field("fans")
@Min(value = 0, message = "Los fans han de ser minimo 0.")
private int fans;

@Field("average_stars")
@Min(value = 0, message = "Las estrellas deben estar entre 0 y 5.")
@Max(value = 5, message = "Las estrellas deben estar entre 0 y 5.")
private double averageStars;

@Field("compliment_hot")
@Min(value = 0, message = "Los complimentHot han de ser minimo 0.")
private int complimentHot;

@Field("compliment_more")
@Min(value = 0, message = "Los complimentMore han de ser minimo 0.")
private int complimentMore;
```

```java
@Field("compliment_profile")
@Min(value = 0, message = "Los complimentProfile han de ser minimo 0.")
private int complimentProfile;

@Field("compliment_cute")
@Min(value = 0, message = "Los complimentCute han de ser minimo 0.")
private int complimentCute;

@Field("compliment_list")
@Min(value = 0, message = "Los complimentList han de ser minimo 0.")
private int complimentList;

@Field("compliment_note")
@Min(value = 0, message = "Los complimentNote han de ser minimo 0.")
private int complimentNote;

@Field("compliment_plain")
@Min(value = 0, message = "Los complimentPlain han de ser minimo 0.")
private int complimentPlain;

@Field("compliment_cool")
@Min(value = 0, message = "Los complimentCool han de ser minimo 0.")
private int complimentCool;

@Field("compliment_funny")
@Min(value = 0, message = "Los complimentFunny han de ser minimo 0.")
private int complimentFunny;
```

```java
@Field("compliment_writer")
@Min(value = 0, message = "Los complimentWriter han de ser minimo 0.")
private int complimentWriter;

@Field("compliment_photos")
@Min(value = 0, message = "Los complimentPhotos han de ser minimo 0.")
private int complimentPhotos;
```

Para sintetizar todo el código de la aplicación a continuación mostramos el código de solo 8 de las operaciones CRUD totales que tiene el proyecto, así como también el código asociado a las acciones obligatorias de la práctica.

Comenzamos con las dos consultas que implican dos o más colecciones por referencia:

1) **Consulta getInvoicesForBusiness(String businessId, LocalDateTime targetDate)**

Este método se ubica en el servicio de la entidad Business ya que va a ser expuesto al usuario a través de una vista con el correspondiente controlador. La consulta hace lo siguiente:

```java
@Transactional
public List<InvoiceDTO> getInvoicesForBusiness(String businessId, LocalDateTime targetDate)
{
    // Etapa 1: Filtrar negocios por business_id
    AggregationOperation matchBusinessId = match(Criteria.where("business_id").is(businessId));

    // Etapa 2: Hacer lookup con la colección "checkin"
    LookupOperation lookupInvoices = LookupOperation.newLookup()
            .from("checkin")
            .localField("business_id")
            .foreignField("business_id")
            .as("invoices");

    // Etapa 3: Desenrollar el array de facturas
    UnwindOperation unwindInvoices = Aggregation.unwind("invoices");

    // Etapa 4: Filtrar facturas por fecha
    AggregationOperation matchByDate = match(Criteria.where("invoices.date").in(targetDate));

    // Etapa 5: Seleccionar los campos relevantes
    AggregationOperation projectFields = project("name", "address")
            .and("invoices").as("invoiceDetails");
```

Primero filtramos los negocios por business_id indicado por parámetro, después hacemos una unión entre las colecciones Checkin y Business con el comando lookup, a continuación, desenrollamos el array creado anteriormente de facturas para posteriormente, filtrar, por la fecha indicada por parámetro. Después seleccionamos los campos relevantes para mostrar su información.

```
// Construir la agregación
Aggregation aggregation = newAggregation(
        matchBusinessId,
        lookupInvoices,
        unwindInvoices,
        matchByDate,
        projectFields
);

// Ejecutar la consulta
AggregationResults<Document> results = mongoTemplate.aggregate(aggregation, "business", Document.class);
/*results.forEach(doc → {
    System.out.println(doc.toJson());
});*/

// Mapeo de los resultados a InvoiceDTO
return results.getMappedResults().stream().map(doc → {
    InvoiceDTO dto = new InvoiceDTO();

    // Crear el DTO para los detalles de la factura
    InvoiceDetailsDTO invoiceDetailsDTO = new InvoiceDetailsDTO();

    // Acceder al campo invoiceDetails que contiene la información de la factura
    Object invoiceDetails = doc.get("invoiceDetails");
```

En esta parte construimos toda la agregación juntando cada parte anterior y ejecutamos la consulta.

```
if (invoiceDetails instanceof Document) {
    Document invoiceDoc = (Document) invoiceDetails;

    // Obtener el ObjectId y convertirlo a String
    ObjectId invoiceId = invoiceDoc.getObjectId("_id");  // Obtienes el ObjectId
    if (invoiceId != null) {
        invoiceDetailsDTO.setInvoiceId(invoiceId.toString());  // Convertir a String
    }

    // Manejar invoiceDates, que puede ser una lista o un solo string
    Object invoiceDates = invoiceDoc.get("date");
    if (invoiceDates instanceof String) {
        // Si es un solo string, conviértelo en una lista con un único elemento
        invoiceDetailsDTO.setInvoiceDates(List.of((String) invoiceDates));
    } else if (invoiceDates instanceof List) {
        // Si ya es una lista, simplemente asígnala
        invoiceDetailsDTO.setInvoiceDates((List<String>) invoiceDates);
    } else {
        // Si es nulo o no esperado, inicializa como lista vacía
        invoiceDetailsDTO.setInvoiceDates(Collections.emptyList());
    }

    // Asignar Business ID (ID del negocio) desde invoiceDetails
    dto.setBusinessId(invoiceDoc.getString("business_id"));
}
```

```
    // Asignar los demás campos desde el documento principal
    dto.setName(doc.getString("name"));
    dto.setAddress(doc.getString("address"));

    // Asignar el DTO de invoiceDetails
    dto.setInvoiceDetails(invoiceDetailsDTO);

    return dto;
}).toList();
```

Para finalizar los resultados, que van a ser documentos de tipos bSON, han de ser transformados a un tipo de datos concreto para poder ser tratados y mostrados, por tanto, creamos las clases Invoices e InvoicesDetails que recoge la información de dichos documentos y para ellos nos valemos del patrón de diseño DTO, que en Spring sería de esta manera:

```
public class InvoiceDTO
{

    private String name;
    private String address;
    private String businessId;
    private InvoiceDetailsDTO invoiceDetails;
```

```
public class InvoiceDetailsDTO
{

    private String invoiceId;
    private List<String> invoiceDates;
```

Estas clases son usadas para mapear los resultados de la consulta, donde asignamos los correspondientes campos y los devolvemos en forma de lista.

2) Consulta getReviewsByUserAndDate(String userId, LocalDateTime date)

Este método se ubica en el servicio de la entidad Review ya que va a ser expuesto al usuario a través de una vista con el correspondiente controlador. La consulta hace lo siguiente:

```
@Transactional
public List<ReviewDTO> getReviewsByUserAndDate(String userId, LocalDateTime date)
{
    // Paso 1: Filtrar reseñas por user_id y fecha
    AggregationOperation matchUserAndDate = match(Criteria.where("user_id").is(userId).and("date").in(date));

    // Paso 2: Lookup para obtener información del negocio
    LookupOperation lookupBusinessDetails = LookupOperation.newLookup()
            .from("business")
            .localField("business_id")
            .foreignField("business_id")
            .as("businessDetails");

    // Paso 3: Desenrollar los detalles del negocio
    UnwindOperation unwindBusinessDetails = unwind("businessDetails");

    // Paso 4: Seleccionar campos relevantes
    AggregationOperation projectFields = Aggregation.project()
            .and("_id").as("reviewId")
            .and("user_id").as("userId")
            .and("business_id").as("businessId")
            .and("businessDetails.name").as("businessName")
            .and("text").as("reviewText")
            .and("stars").as("stars")
            .and("date").as("date");
```

En primer lugar, filtramos por el id del usuario y la fecha indicadas por parámetro, después, con el comando lookup juntamos las colecciones Review y User. Posteriormente, desenrollamos los detalles de esta unión para poder después seleccionar los campos relevantes que deseamos mostrar.

```
Aggregation aggregation = newAggregation(
        matchUserAndDate,
        lookupBusinessDetails,
        unwindBusinessDetails,
        projectFields
);

// Ejecución de la consulta
AggregationResults<Document> results = mongoTemplate.aggregate(aggregation, "review", Document.class);

// Mapear los resultados a DTOs
return results.getMappedResults().stream().map(doc → {
    ReviewDTO dto = new ReviewDTO();
    dto.setReviewId(doc.getObjectId("reviewId").toString());
    dto.setUserId(doc.getString("userId"));
    dto.setBusinessId(doc.getString("businessId"));
    dto.setBusinessName(doc.getString("businessName"));
    dto.setReviewText(doc.getString("reviewText"));
    dto.setStars(doc.getInteger("stars"));
    String dateString = doc.getString("date");
    if (dateString ≠ null) {
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
        dto.setDate(LocalDateTime.parse(dateString, formatter));
    } else {
        dto.setDate(null); // O un valor por defecto
    }

    return dto;
}).toList();
```

En esta parte, construimos toda la agregación con los pasos anteriores y ejecutamos la consulta. Como sucedió con la consulta anterior en necesario crear un objeto DTO para poder mapear los resultados que arroja Mongo, por lo que hemos creado la clase ReviewDTO:

```
public class ReviewDTO
{

    private String reviewId;
    private String userId;
    private String businessId;
    private String businessName;
    private String reviewText;
    private int stars;
    private LocalDateTime date;
```

Desde la cual mapeamos los resultados y los devolvemos en una lista de dichos objetos asignándoles previamente a cada uno sus respectivos campos necesarios.

Seguimos con las dos consultas sobre el campo text de la colección Review:

3) Consulta searchReviewsByKeyword(String keyword, int page, int size)

Este método se ubica en el servicio de la entidad Review ya que va a ser expuesto al usuario a través de una vista con el correspondiente controlador. La consulta hace lo siguiente:

```
public Page<Review> searchReviewsByKeyword(String keyword, int page, int size)
{
    Pageable pageable = PageRequest.of(page, size);
    return reviewRepository.findByTextContainingKeyword(keyword, pageable);
}
```

Se crea un objeto de tipo Pageable, que nos permite manejar la paginación de los resultados aportados por Mongo cuando se ejecute la consulta en base a los parámetros *page* y *size,* para después, llamar al método del repositorio findByTextContainingKeyword, pasándole la palabra clave por la que queremos buscar y el objeto pageable. El motivo del uso del tipo Pageable y del tipo de Java se explicará en el apartado del análisis del rendimiento.

```
// Consulta 1: Buscar palabras clave en el campo 'text' con paginación
@Query("{ $text: { $search: ?0 } }")
Page<Review> findByTextContainingKeyword(String keyword, Pageable pageable);
```

Este método del repositorio de Review esta anotado con @Query para definir la consulta que se ejecuta en MongoDB, que, en este caso, simplemente hace una búsqueda usando un índice de texto sobre el campo text de la colección buscando por la palabra clave que se le pasa por parámetro.

4) Consulta searchReviewsByKeywords(String keywords, int page, int size)

Este método se ubica en el servicio de la entidad Review ya que va a ser expuesto al usuario a través de una vista con el correspondiente controlador. La consulta hace lo siguiente:

```
public Page<Review> searchReviewsByKeywords(String keywords, int page, int size)
{
    Pageable pageable = PageRequest.of(page, size);
    return reviewRepository.findByTextWithKeywordsSortedByRelevance(keywords, pageable);
}
```

Se crea un objeto de tipo Pageable, que nos permite manejar la paginación de los resultados aportados por Mongo cuando se ejecute la consulta en base a los parámetros *page* y *size,* para después, llamar al método del repositorio findByTextWithKeywordSortedByRelevance, pasándole las palabras claves por la que

queremos buscar y el objeto pageable. El motivo del uso del tipo Pageable y del tipo de Java se explicará en el apartado del análisis del rendimiento.

```
// Consulta 2: Buscar con múltiples palabras clave ordenadas por relevancia con paginación
@Query(value = "{ $text: { $search: ?0 } }", sort = "{ score: { $meta: 'textScore' } }")
Page<Review> findByTextWithKeywordsSortedByRelevance(String keywords, Pageable pageable);
```

Este método del repositorio de Review esta anotado con @Query para definir la consulta que se ejecuta en MongoDB, que, en este caso, además de realizar una búsqueda sobre el campo text usando un índice de texto, ordena los resultados basándonos en su relevancia, la cual es calculada por Mongo mediante el textScore para cada documento según su correspondencia con las palabras claves.

Pasamos ahora con las dos consultas usando el Framework Aggregation de Mongo:

5) Consulta getTopRatedBusinessesByCategory(String category)

Este método se ubica en el servicio de la entidad Business ya que va a ser expuesto al usuario a través de una vista con el correspondiente controlador. La consulta hace lo siguiente:

```
public List<Business> getTopRatedBusinessesByCategory(String category)
{
    Aggregation aggregation = Aggregation.newAggregation(
            Aggregation.match(Criteria.where("categories").is(category)), // Filtra por categoría
            Aggregation.sort(Sort.by(Sort.Order.desc("stars"))), // Ordena por la calificación (stars)
            Aggregation.limit(10) // Limita a los 10 primeros negocios
    );

    AggregationResults<Business> results = mongoTemplate.aggregate(aggregation, "business", Business.class);
    return results.getMappedResults();
}
```

De forma general el método realiza una agregación en la colección business de MongoDB para filtrar negocios por categoría, ordenarlos por calificación de estrellas en orden descendente, y devolver los 10 mejores resultados. La agregación en MongoDB está compuesta de varias etapas, estas etapas se especifican como una secuencia de operaciones en el pipeline.

En la primera etapa filtra los documentos de la colección business para incluir solo aquellos en los que el campo categories coincida con el valor proporcionado en el parámetro category. Internamente utiliza el operador $match de MongoDB.

En la segunda etapa, ordena los documentos por el campo stars en orden descendente y para ello utiliza el operador $sort de MongoDB. Por último, se limita el resultado a los primeros 10 documentos, para ello utiliza el operador $limit.

Por último, se ejecuta la agregación en la base de datos con la agregación definida anteriormente, el nombre de la colección donde se va a ejecutar y la clase Java a la que se mapean los documentos resultantes, devolviendo una lista de dichos objetos.

6) Consulta getBusinessesWithMostReviewsByCategory(String category)

Este método se ubica en el servicio de la entidad Business ya que va a ser expuesto al usuario a través de una vista con el correspondiente controlador. La consulta hace lo siguiente:

```
public List<Business> getBusinessesWithMostReviewsByCategory(String category)
{
    Aggregation aggregation = Aggregation.newAggregation(
            Aggregation.match(Criteria.where("categories").in(category)), // Filtra por la categoría
            Aggregation.sort(Sort.by(Sort.Order.desc("review_count"))), // Ordena por el número de reseñas
            Aggregation.limit(10) // Limita a los 10 negocios con más reseñas
    );

    AggregationResults<Business> results = mongoTemplate.aggregate(aggregation, "business", Business.class);
    return results.getMappedResults();
}
```

Este método obtiene los 10 negocios con más reseñas dentro de una categoría específica. Se utiliza la clase mongoTemplate de Spring Data MongoDB para ejecutar una consulta de agregación. La agregación en MongoDB está compuesta de varias etapas, estas etapas se especifican como una secuencia de operaciones en el pipeline.

En la primera etapa, filtra los documentos que contienen la categoría especificada en el campo categories. Internamente usa el operador $match de MongoDB para realizar esta operación. La condición in(category) asegura que el negocio esté asociado con la categoría dada.

En la segunda etapa, ordena los documentos por el campo review_count en orden descendente. Usa el operador $sort de MongoDB para realizar esta operación. Este paso asegura que los negocios con más reseñas aparezcan primero.

Por último, se limitan los resultados a los primeros 10 documentos usando el operador $limit. Después se ejecuta el pipeline de agregación en la base de datos, con los pasos de agregación, el nombre de la colección sobre la cual se va a ejecutar y la clase java en la que se mapearán los documentos resultantes, devolviendo una lista de objetos de este tipo.

7) Consulta findByNameContainingIgnoreCase(String name, Pageable pageable)

Este método define una consulta personalizada en el repositorio de la entidad User de Spring Data MongoDB para buscar usuarios por nombre. Hace lo siguiente:

```
@Query("{'name' : ?0}")
Page<User> findByNameContainingIgnoreCase(String name, Pageable pageable);
```

El método busca documentos en la colección que contienen un campo name coincidente con el parámetro proporcionado. Por último, devuelve los resultados en un formato de página (Page<User>), limitado al tamaño y orden especificado en pageable.

8) **Consulta findByUserIdContainingIgnoreCase(String userId, Pageable pageable)**

El método busca las reseñas (Review) cuyo campo user_id coincida con el valor del parámetro userId.

```
@Query("{'user_id' : ?0}")
Page<Review> findByUserIdContainingIgnoreCase(String userId, Pageable pageable);
```

Busca documentos donde el valor del campo user_id coincida exactamente con el valor proporcionado como parámetro. Los resultados se devuelven como una página (Page<Review>), lo que permite paginar los datos en conjuntos más manejables.

Análisis del rendimiento de consultas

En este apartado vamos a analizar el rendimiento de 3 de las consultas propuestas en el proyecto. En este caso analizaremos las consultas, getTopRatedBusinessesByCategory(String category), getInvoicesForBusiness(String businessId, LocalDateTime targetDate) y findByNameContainingIgnoreCase(String name, Pageable pageable).

Para cada consulta veremos su rendimiento original y propondremos distintas estrategias para su optimización, obviamente ya implementada, la que dé mejor resultado, en el código aportado de la práctica.

El primer lugar, analizaremos la consulta getTopRatedBusinessesByCategory(String category), para ello ejecutaremos el explain de la consulta con un ejemplo para ver su rendimiento:

```
db.business.explain("executionStats").aggregate([
    { $match: { categories: "Restaurant" } },
    { $sort: { stars: -1 } },
    { $limit: 10 }
]);
```

Cuyos resultados son los siguientes:

```
winningPlan: {
  isCached: false,
  stage: 'SORT',
  sortPattern: {
    stars: -1
  },
  memLimit: 104857600,
  limitAmount: 10,
  type: 'simple',
  inputStage: {
    stage: 'COLLSCAN',
    filter: {
      categories: {
        '$eq': 'Restaurant'
      }
    },
    direction: 'forward'
  }
},
rejectedPlans: []
```

```
executionStats: {
  executionSuccess: true,
  nReturned: 0,
  executionTimeMillis: 292,
  totalKeysExamined: 0,
  totalDocsExamined: 150346,
  executionStages: {
    isCached: false,
    stage: 'SORT',
    nReturned: 0,
    executionTimeMillisEstimate: 289,
    works: 150348,
    advanced: 0,
    needTime: 150347,
    needYield: 0,
    saveState: 19,
    restoreState: 19,
    isEOF: 1,
    sortPattern: {
      stars: -1
    },
```

Identificamos varios problemas:

- El filtro categories no usa ningún índice, esto obliga MondoDB a revisar cada documento, lo que escala mal con gradnes volúmenes de datos.
- No se valora ningún otro plan, debido a la falta de índices.
- Escaneo completo de la colección (COLLSCAN)
- Tiempo de ejecución de 292ms, relativamente alto debido al escaneo completo.

Por tanto, vemos claramente que para mejorar el rendimiento necesitamos la creación de índices y proponemos lo siguiente:

- Creación de un índice compuesto de los campos categories y stars de la colección business:

```
db.business.createIndex({ categories: 1, stars: -1 });
```

  Lo cual nos dará la ventaja de que MongoDB podrá filtrar por categories y ordenar por stars sin necesidad de escanear la colección ni ordenar en memoria.

- Si ciertas categorías son consultadas con más frecuencia, se pueden usar índices parciales para optimizar estas búsquedas. Esta técnica no se ha implementado porque se tendría que evaluar cuales son las categorías más consultadas para evitar hacer índices parciales en todas las categorías disponibles.
- Si el campo categories contiene múltiples valores como un array, evaluar si es mejor mantener una colección separada de categorías para optimizar las búsquedas.

Volviendo a ejecutar el mismo explain y obtenemos los siguientes resultados:

```
winningPlan: {
  isCached: false,
  stage: 'LIMIT',
  limitAmount: 10,
  inputStage: {
    stage: 'FETCH',
    inputStage: {
      stage: 'IXSCAN',
      keyPattern: {
        categories: 1,
        stars: -1
      },
      indexName: 'categories_1_stars_-1',
      isMultiKey: true,
      multiKeyPaths: {
        categories: [
          'categories'
        ],
        stars: []
      },
```

```
executionStats: {
  executionSuccess: true,
  nReturned: 0,
  executionTimeMillis: 0,
  totalKeysExamined: 0,
  totalDocsExamined: 0,
  executionStages: {
    isCached: false,
    stage: 'LIMIT',
    nReturned: 0,
    executionTimeMillisEstimate: 0,
    works: 1,
    advanced: 0,
    needTime: 0,
    needYield: 0,
    saveState: 0,
    restoreState: 0,
    isEOF: 1,
    limitAmount: 10,
```

Observamos una mejora del rendimiento que vamos a representar en la siguiente tabla:

Aspecto	Plan original	Plan optimizado
Escaneo de la colección	COLLSCAN: 150.346 documentos	IXSCAN: 0 documentos
Ordenamiento SORT	En memoria, potencialmente costoso	Evitado, gracias al índice
Documentos examinados	150.346	0
Tiempo de ejecución	292 ms	0 ms

Pasemos a analizar la segunda consulta, getInvoicesForBusiness(String businessId, LocalDateTime targetDate)), para ello ejecutaremos el explain de la consulta con un ejemplo para ver su rendimiento:

```
db.business.explain("allPlansExecution").aggregate([
    { $match: { business_id: "Pns2l4eNsfO8kk83dixA6A" } },
    { $lookup: {
        from: "checkin",
        localField: "business_id",
        foreignField: "business_id",
        as: "invoices"
    }},
    { $unwind: "$invoices" },
    { $match: { "invoices.date": { $in: ["2018-09-21 20:51:31"] } } },
    { $project: { name: 1, address: 1, invoices: 1 }}
])
```

Identificamos varios problemas:

- El resultado del explain() indica que la consulta está utilizando un COLLSCAN (un escaneo completo de la colección) en la primera fase de la agregación, lo cual es ineficiente, especialmente si la colección business es grande. Este escaneo completo ocurre porque la consulta está filtrando por el campo business_id, pero no se está utilizando ningún índice para optimizar este filtro.
- Algo similar ocurre para el campo date de la colección checkin, que para ayudar al filtro del match y acelerar el lookup se debería crear un índice sobre dicho campo también

Procedemos a continuación a implementar dichas propuestas:

- Crear un índice compuesto en los campos business_id y invoices.date para mejorar el filtrado inicial.

```
yelp> db.checkin.createIndex({ business_id: 1, "date": 1 });
```

- Crear un índice sobre el campo business_id de la colección Business.

```
yelp> db.business.createIndex({ business_id: 1 });
```

Si volvemos a ejecutar el mismo explain() para comprobar si ha habido mejora de rendimiento:

```
winningPlan: {
  isCached: false,
  stage: 'PROJECTION_SIMPLE',
  transformBy: {
    _id: 1,
    address: 1,
    business_id: 1,
    name: 1
  },
  inputStage: {
    stage: 'FETCH',
    inputStage: {
      stage: 'IXSCAN',
      keyPattern: {
        business_id: 1
      },
      indexName: 'business_id_1',
      isMultiKey: false,
      multiKeyPaths: {
        business_id: []
      },
```

```
executionStats: {
  executionSuccess: true,
  nReturned: 1,
  executionTimeMillis: 0,
  totalKeysExamined: 1,
  totalDocsExamined: 1,
  executionStages: {
    isCached: false,
    stage: 'PROJECTION_SIMPLE',
    nReturned: 1,
    executionTimeMillisEstimate: 0,
    works: 2,
    advanced: 1,
    needTime: 0,
    needYield: 0,
    saveState: 1,
    restoreState: 1,
    isEOF: 1,
    transformBy: {
      _id: 1,
      address: 1,
      business_id: 1,
      name: 1
    },
```

Observamos una mejora en el rendimiento que representamos en esta tabla:

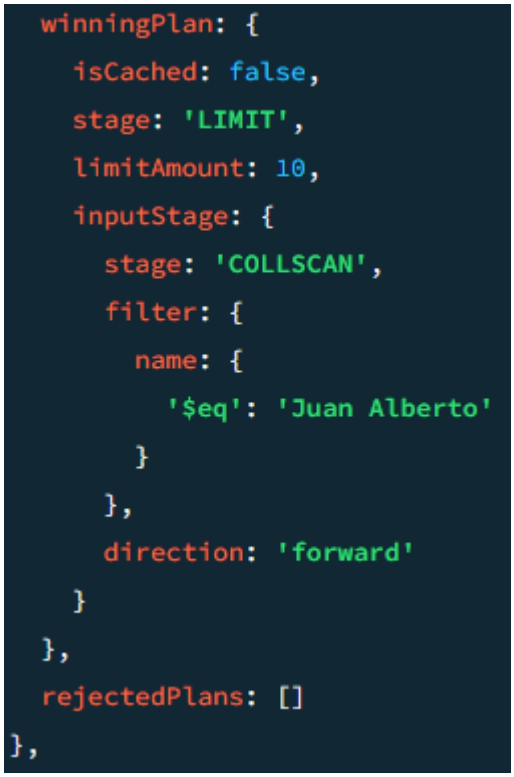
Aspecto	Plan original	Plan optimizado
Escaneo de la colección	COLLSCAN: 150.346 documentos	IXSCAN: 1 documentos
Uso de Índices	No se utilizó ningún índice (indexesUsed: []).	Se utilizó el índice business_id_1 para la consulta en business_id y el índice business_id_1_date_1 para el lookup
Documentos examinados	150.346	1 documento fue examinado, ya que el índice ayudó a limitar la búsqueda.
Tiempo de ejecución	87 ms	0 ms

Por último, vamos a pasar a analizar la tercera consulta que es findByNameContainingIgnoreCase(String name, Pageable pageable), para ello al igual que con las otras consultas vamos a lanzar su explain en mongo:

```
db.user.explain("allPlansExecution").find({ name: "Juan Alberto" }).limit(10)
```

Identificamos los siguientes problemas:

- La consulta escanea completamente toda la colección (COLLSCAN), por lo que MongoDB revisó todos los documentos de la colección user para encontrar coincidencias, porque no había un índice adecuado en el campo name.
- Tiempo de ejecución de 22 segundos, un tiempo muy alto para una consulta que devuelve un único resultado.



```
  winningPlan: {
    isCached: false,
    stage: 'LIMIT',
    limitAmount: 10,
    inputStage: {
      stage: 'COLLSCAN',
      filter: {
        name: {
          '$eq': 'Juan Alberto'
        }
      },
      direction: 'forward'
    }
  },
  rejectedPlans: []
},
```

```
executionStats: {
  executionSuccess: true,
  nReturned: 1,
  executionTimeMillis: 21920,
  totalKeysExamined: 0,
  totalDocsExamined: 1987897,
  executionStages: {
    isCached: false,
    stage: 'LIMIT',
    nReturned: 1,
    executionTimeMillisEstimate: 21697,
    works: 1987898,
    advanced: 1,
    needTime: 1987896,
    needYield: 0,
    saveState: 1312,
    restoreState: 1312,
    isEOF: 1,
    limitAmount: 10,
    inputStage: {
      stage: 'COLLSCAN',
      filter: {
        name: {
          '$eq': 'Juan Alberto'
        }
      },
      nReturned: 1,
      executionTimeMillisEstimate: 21675,
      works: 1987898,
```

Está claro que lo necesario para mejorar el rendimiento de está el crear un índice para el campo name de la colección User:

```
yelp> db.user.createIndex({ name: 1 })
```

Si volvemos a ejecutar el mismo explain sobre la consulta, veremos si se produce mejora en el rendimiento o no:

```
winningPlan: {
  isCached: false,
  stage: 'LIMIT',
  limitAmount: 10,
  inputStage: {
    stage: 'FETCH',
    inputStage: {
      stage: 'IXSCAN',
      keyPattern: {
        name: 1
      },
      indexName: 'name_1',
      isMultiKey: false,
      multiKeyPaths: {
        name: []
      },
      isUnique: false,
      isSparse: false,
      isPartial: false,
      indexVersion: 2,
      direction: 'forward',
      indexBounds: {
        name: [
          '["Juan Alberto", "Juan Alberto"]'
        ]
      }
    }
  }
```

```
executionStats: {
  executionSuccess: true,
  nReturned: 1,
  executionTimeMillis: 0,
  totalKeysExamined: 1,
  totalDocsExamined: 1,
  executionStages: {
    isCached: false,
    stage: 'LIMIT',
    nReturned: 1,
    executionTimeMillisEstimate: 0,
    works: 2,
    advanced: 1,
    needTime: 0,
    needYield: 0,
    saveState: 0,
    restoreState: 0,
    isEOF: 1,
    limitAmount: 10,
```

Efectivamente el rendimiento mejora sensiblemente tal y como recogemos en la siguiente tabla:

Aspecto	Plan original	Plan optimizado
Escaneo de la colección	COLLSCAN: 1.987.897 documentos	IXSCAN: 1 documentos
Uso de Índices	No se utilizó ningún índice (indexesUsed: []).	Se utilizó el índice name_1 para la consulta
Documentos examinados	1.987.897	1 documento fue examinado, ya que el índice ayudó a limitar la búsqueda.
Tiempo de ejecución	22.000 ms	0 ms

Conclusiones

Como conclusiones podemos decir que este proyecto me ha servido para integrar una base de datos Mongo con el framework de Spring y una aplicación web basada en arquitectura cliente-servidor para ofrecer funcionalidad y mostrar datos.

Por otra parte, también he aprendido a optimizar consultas con Mongo para mejorar el rendimiento y el tiempo de respuesta ante las consultas de los usuarios en un entorno con un volumen de datos muy grande y cercano a lo que se podría manejar en una empresa real de cierto tamaño.

Considero la asignatura, como una parte importante del máster por el uso y aprendizaje de un sistema gestor de base de datos NoSQL, muy usado en la actualidad y que suele manejar gran cantidad de datos, lo cuál lo hace interesante para aplicaciones que manejen Big Data, que es la especialidad del máster que me hayo cursando y por tanto, guarda especial relación con ella.