**DOCUMENTOS EN SPRING BOOT**

| **Información general** | |
| --- | --- |
| **Duración estimada en minutos:** | 120 |
| **Docente:** | Carlos Andrés Florez Villarraga |
| **Guía no.** | 06 |

| **Información de la Guía** |
| --- |

**OBJETIVO**

Comprender qué es una entidad en Spring Boot y su equivalencia al concepto de colección en MongoDB.

**CONCEPTOS BÁSICOS**

NoSQL, archivos JSON, programación orientada a objetos, Java.

**CONTEXTUALIZACIÓN TEÓRICA**

Una entidad es un elemento de persistencia, que permite modelar o representar un objeto o información del sistema que estamos construyendo. Toda entidad está compuesta por uno o más atributos, cada uno de los cuales representa una propiedad o información del objeto que se está modelando.

En Spring Boot, cuando se trabaja con persistencia de datos, es común utilizar entidades y documentos para representar objetos que se almacenan en una base de datos. Sin embargo, la elección entre entidad y documento depende del tipo de base de datos que se está utilizando y de la naturaleza de los datos.

Entidades (Entity):

* Las entidades son típicamente utilizadas con bases de datos relacionales (como MySQL, PostgreSQL, etc.).
* Una entidad en Spring Boot generalmente se mapea a una tabla en la base de datos relacional.
* Se pueden utilizar anotaciones de JPA (Java Persistence API) como @Entity, @Table, @Id, etc., para configurar la relación entre la clase y la tabla de la base de datos. Así como anotaciones para definir las relaciones entre entidades como por ejemplo: @ManyToMany, @ManyToOne, @OneToOne, @OneToMany.

Documentos (Document):

* Los documentos son comúnmente utilizados con bases de datos NoSQL, como MongoDB.
* Un documento es una representación de datos en un formato no tabular (como JSON en el caso de MongoDB).
* Se pueden utilizar anotaciones de Spring Data MongoDB, como @Document, para indicar que una clase debe ser almacenada como un documento en MongoDB.

Dado que estamos usando MongoDB como sistema de base de datos, entonces debemos usar Documentos, un documento tiene la siguiente estructura:

| import org.springframework.data.annotation.Id;  import org.springframework.data.mongodb.core.mapping.Document;  @Document  public class Usuario {    @Id  private String id;    private String nombre;  private String email;    // Getters y setters  } |
| --- |

@Document: Es la anotación principal que marca la clase como un documento de MongoDB. Al usar esta anotación, estamos indicando a Spring Data MongoDB que la clase debe ser tratada como un documento y mapeada a una colección de MongoDB. Esto significa que cada instancia de esta clase se almacenará como un documento en la base de datos.

Nombre de la Colección (opcional): Es posible especificar el nombre de la colección en MongoDB donde se almacenarán los documentos de esta clase. Si no se proporciona un nombre, Spring Data MongoDB utilizará el nombre de la clase en minúsculas como nombre de la colección por defecto. Para asignar el nombre se debe usar la propiedad collection, o simplemente escribiendo el nombre entre los paréntesis, ejemplo: @Document(collection = "usuarios")

@Id: Esta anotación se utiliza para marcar un campo como la clave primaria del documento. En el ejemplo anterior, el campo id de la clase Usuario se designa como la clave primaria. Spring Data MongoDB utiliza este campo para asociar un identificador único a cada documento almacenado en la colección MongoDB. El tipo de datos del campo anotado con @Id puede variar (por ejemplo, String, ObjectId, BigInteger, etc.). Spring Data MongoDB maneja automáticamente la generación y asignación del valor de la clave primaria según sea necesario.

Los demás atributos del documento pueden ser los típicos que usamos en Java como números, cadenas de texto, booleanos, arreglos, colecciones, etc.

**PRECAUCIONES Y RECOMENDACIONES**

Recuerde verificar que tiene instalado el JDK de Java (preferiblemente la versión 21). Asegúrese de que tiene el servidor de MongoDB (mongod) en ejecución.

**ARTEFACTOS**

Se requiere tener instalado IntelliJ IDEA en su versión Ultimate, así como MongoDB y el controlador de versiones GIT.

**EVALUACIÓN O RESULTADO**

Se espera que el estudiante logre modelar entidades lógicas de una aplicación y que por medio de ellas se generen las colecciones correspondientes en la base de datos que las representen de forma adecuada.

| **Procedimiento** |
| --- |

1. En esta guía usaremos el ejemplo de la tienda visto anteriormente en la guía de NoSQL. Agregue la siguiente dependencia en el archivo build.gradle.kts.

| *implementation*("org.springframework.boot:spring-boot-starter-data-mongodb") |
| --- |

Sincronice Gradle para que descargue la dependencia.

1. Escriba lo siguiente en el archivo application.properties archivo:

| spring.data.mongodb.uri=mongodb://localhost:27017/proyecto  logging.level.org.springframework.data.mongodb.core.MongoTemplate=DEBUG |
| --- |

El archivo application.properties es fundamental en un proyecto de Spring. En este archivo se define la configuración de una aplicación, como por ejemplo, la conexión a la base de datos, el uso de modo debug para hacer un logging más detallado, entre otras cosas. Para conocer más sobre las propiedades que pueden ir en este archivo, podemos visitar:

<https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html>

Para nuestro caso, el archivo de propiedades, tiene los datos de conexión a la base de datos de MongoDB (sin autenticación). Es decir, nos vamos a conectar a la base de datos con el nombre proyecto que está en el puerto 27017 (que es el puerto por defecto de MongoDB) en localhost.

1. Ubíquese en la carpeta main/java y cree allí el paquete: co.edu.uniquindio.proyecto.modelo. En dicho paquete debe crear la clase Cliente. Para este caso, la clase Cliente tiene los atributos: cedula, nombre, email, telefono.

| package co.edu.uniquindio.proyecto.modelo;  import org.bson.types.ObjectId;  import org.springframework.data.annotation.Id;  import org.springframework.data.mongodb.core.mapping.Document;  @Document("clientes")  public class Cliente {  @Id  private ObjectId codigo;    private String cedula;  private String nombre;  private String email;  private String telefono;    } |
| --- |

Tenga en cuenta que se creó un atributo nuevo que se llama codigo como “llave primaria”, ya que tiene la anotación @Id. Este código es generado de forma automática por MongoDB, por lo tanto no nos debemos preocupar por él, todos los documentos deben tener este mismo atributo.

**NOTA:** Cambie proyecto por el nombre de su proyecto (o el nombre del paquete que ya haya creado previamente (así mismo con el resto de la guía).

1. En Java es muy común agregar Getters y Setters, constructores, hashCode(), equals(), toString(), y esto implica que el código de las clases se vea extenso, repetitivo e incluso molesto. Para solucionar en parte esto, existe una librería que se llama lombok: <https://projectlombok.org/>. Vamos a usarla en el proyecto.

Si los tiene, borre todos los métodos get() y set(), así como el toString() y agregue las anotaciones @Getter, @Setter y @ToString a la clase.

1. También se puede usar la anotación @ToString, @EqualsAndHashCode. Para estos dos casos tenga en cuenta que por defecto se considerarán todos los atributos de la clase para generar dichos métodos. Esto puede ser un problema ya que por ejemplo en el @EqualsAndHashCode solo se debe considerar la llave primaria de la clase (o el id), entonces para arreglar esto podemos usar la anotación Include o exclude. Si se desea usar el Include, debe añadir la propiedad: onlyExplicitlyIncluded.

Para más información visitar:

* <https://www.paradigmadigital.com/dev/proyecto-lombok-facilitame-la-vida/>

Con los ajustes indicados previamente, la clase Cliente debe quedar así:

| package co.edu.uniquindio.proyecto.modelo;  import org.bson.types.ObjectId;  import org.springframework.data.annotation.Id;  import org.springframework.data.mongodb.core.mapping.Document;  import lombok.\*;  @Document("clientes")  @Getter  @NoArgsConstructor  @ToString  @EqualsAndHashCode(onlyExplicitlyIncluded = true)  public class Cliente {    @Id  @EqualsAndHashCode.Include  private ObjectId codigo;  private String cedula;  private String nombre;  private String email;  private String telefono;  } |
| --- |

1. Modifique la clase Cliente para que en lugar de tener un sólo teléfono tenga varios. Para ello use un List.

| private List<String> telefonos; |
| --- |

1. Cree la clase Producto de la siguiente manera:

| package co.edu.uniquindio.proyecto.modelo;  import org.bson.types.ObjectId;  import org.springframework.data.annotation.Id;  import org.springframework.data.mongodb.core.mapping.Document;  import lombok.\*;  @Document("productos")  @Getter  @Setter  @NoArgsConstructor  @ToString  @EqualsAndHashCode(onlyExplicitlyIncluded = true)  public class Producto {  @Id  @EqualsAndHashCode.Include  private ObjectId codigo;  private String nombre;  private TipoProducto tipoProducto;  private int unidades;  private float precio;  } |
| --- |

1. Cree una enumeración que le permita representar los diferentes tipos de productos. El nombre del enum debe ser TipoProducto. Agregue algunos valores al enum.
2. Si lo desea puede hacer que en lugar de que el producto tenga un solo tipo, tenga varios. Para esto solo debe hacer lo mismo que se hizo con los teléfonos del cliente. cambiar TipoProducto a List<TipoProducto>.
3. Cree las clases que representan el Pago y la Transacción. Recuerde generar el hashCode, equals, así como los getters, setter y toString con Lombok.

La clase Transaccion puede quedar así:

| package co.edu.uniquindio.proyecto.modelo;  import lombok.\*;  import org.springframework.data.annotation.Id;  import org.springframework.data.mongodb.core.mapping.Document;  import java.time.LocalDateTime;  import java.util.List;  import org.bson.types.ObjectId;  @Document("transacciones")  @Getter  @Setter  @NoArgsConstructor  @ToString  @EqualsAndHashCode(onlyExplicitlyIncluded = true)  public class Transaccion {  @Id  @EqualsAndHashCode.Include  private ObjectId codigo;  private List<ObjectId> productos;  private ObjectId idCliente;  private LocalDateTime fecha;  private Pago pago;  } |
| --- |

Observe que el atributo idCliente es de tipo ObjectId, ya que realmente no se guarda todo el Cliente en la transacción sino su id (que es de tipo ObjectId en MongoDB). Igualmente tenemos una lista de productos de tipo ObjectId, ya que en dicha lista guardamos los id de cada producto y no el Producto completo. Esto se hace así para reducir la redundancia entre las colecciones.

La transacción tiene un atributo que representa el pago, en este caso el pago si es un objeto ya que viene de la clase Pago. A continuación se muestra la clase Pago y sus atributos:

| package co.edu.uniquindio.proyecto.modelo;  import java.time.LocalDateTime;  import lombok.AllArgsConstructor;  import lombok.Getter;  import lombok.Setter;  @AllArgsConstructor  @Getter  @Setter  public class Pago {  private LocalDateTime fecha;  private float totalPagado;  private String estado;  private String metodoPago;  } |
| --- |

Pero esta clase no es un @Document, es decir, no se convertirá en una colección propia en MongoDB sino que hará parte de la colección transacciones. Esto se hace así ya que el pago depende exclusivamente de una transacción. Sin una transacción el pago no tendría sentido, por lo tanto tener una colección para solo los pagos no es recomendable.

Así se verá una transacción con dos productos guardada en MongoDB siguiendo este esquema de clases:

| {  \_id: ObjectId('65b16f5f84e34686d4803708'),  idCliente: ObjectId('663eb04a4af4c9d1c336ebb2'),  productos: [ObjectId('663eb0e378e63f2ed0113b4d'), ObjectId('663eb0eb047e23deb49ebdc0')],  fecha: ISODate('2024-01-09T12:00:00.000Z'),  pago: {  fecha: ISODate('2024-01-09T12:30:00.000Z'),  totalPagado: 2400,  estado: 'Pagado',  metodoPago: 'Tarjeta de crédito'  }  } |
| --- |

1. Un ajuste que es importante tener en cuenta en esta implementación es que en una transacción pueden haber diferentes unidades de cada producto y es necesario que quede registro del precio unitario de cada producto al momento de hacer la transacción.

Para esto, basta con crear una clase que se llame DetalleProducto y modificar la clase Transaccion para que ya no sea un List<ObjectId> sino un List<DetalleProducto> para los productos. A continuación se muestra la clase DetalleProducto:

| package co.edu.uniquindio.proyecto.modelo;  import lombok.AllArgsConstructor;  import lombok.Getter;  import lombok.Setter;  import org.bson.types.ObjectId;  @AllArgsConstructor  @Getter  @Setter  public class DetalleProducto {    private ObjectId idProducto;  private float precio;  private int cantidad;  } |
| --- |

Con este ajuste, así se verá una transacción con dos productos guardada en MongoDB siguiendo el esquema de clases:

| {  \_id: ObjectId('65b16f5f84e34686d4803708'),  idCliente: ObjectId('663eb04a4af4c9d1c336ebb2'),  productos: [  {  idProducto: ObjectId('663eb0e378e63f2ed0113b4d'),  cantidad: 2,  precio: 160000,  },  {  idProducto: ObjectId('663eb0eb047e23deb49ebdc0'),  cantidad: 1,  precio: 58000,  },  ],  fecha: ISODate('2024-01-09T12:00:00.000Z'),  pago: {  fecha: ISODate('2024-01-09T12:30:00.000Z'),  totalPagado: 2400,  estado: 'Pagado',  metodoPago: 'Tarjeta de crédito'  }  } |
| --- |

1. Para cada clase del modelo, genere un constructor que incluya todos los atributos, excepto aquel marcado con @Id. Aplique la anotación @Builder al constructor, ya que esta facilita la creación de objetos mediante el uso del patrón de diseño Builder, que permite construir instancias de manera más sencilla y flexible.

Más información: <https://projectlombok.org/features/Builder>

1. (**ESTE PUNTO ES OPCIONAL**) La decisión de tener los pagos en una colección aparte o incluirlos como parte de cada transacción depende de varios factores y consideraciones específicas de la aplicación que estemos desarrollando.

Aquí hay algunos aspectos a tener en cuenta que podrían influir en la elección:

* Naturaleza de los Datos: Si los pagos tienen una entidad independiente y pueden existir por sí mismos, es posible que sea más apropiado tenerlos en una colección aparte. Esto sería especialmente relevante si los pagos pueden estar asociados con diferentes tipos de entidades en el sistema, no solo transacciones.
* Consultas y Desempeño: Si se espera realizar consultas frecuentes que involucren tanto transacciones como pagos, tener los pagos embebidos en las transacciones puede mejorar el rendimiento. Esto es especialmente cierto si los pagos son simplemente un subconjunto de datos dentro de las transacciones y se accede a ellos juntos con frecuencia.
* Escalabilidad: Si se anticipa que la cantidad de pagos o transacciones en el sistema será grande, se debe considerar la escalabilidad. Tener los pagos en una colección aparte puede facilitar la escalabilidad horizontal.
* Consistencia y Atomicidad: Si se necesita garantizar la consistencia y la atomicidad en la actualización de pagos y transacciones, embeber los pagos en las transacciones puede ser más adecuado, ya que ambas entidades se pueden actualizar dentro de una única operación atómica.
* Requerimientos de Auditoría y Seguimiento: Si se necesita realizar un seguimiento detallado de los cambios en los pagos independientemente de las transacciones, tenerlos en una colección aparte facilita el seguimiento de auditoría.

Si se requiere tener el pago como colección entonces la clase Pago quedaría así:

| package co.edu.uniquindio.proyecto.modelo;  import lombok.AllArgsConstructor;  import lombok.Getter;  import lombok.Setter;  import org.springframework.data.annotation.Id;  import org.springframework.data.mongodb.core.mapping.Document;  import java.time.LocalDateTime;  import org.bson.types.ObjectId;  @Document("pagos")  @AllArgsConstructor  @Getter  @Setter  public class Pago {    @Id  private String codigo;  private LocalDateTime fecha;  private float totalPagado;  private String estado;  private String metodoPago;  private ObjectId idTransaccion;  } |
| --- |

Y a la clase Transaccion se le quita el atributo del pago ya que el Pago tiene el id de la transacción a la que pertenece.

1. Recuerde agregar los archivos modificados y nuevos al repositorio de git y hacer commit.
2. Más adelante veremos cómo le podemos agregar restricciones a los atributos para que no sean nulos o que tengan un tamaño máximo, mínimo, etc.

| **Para la próxima clase** |
| --- |

* Al final de la guía 05 (bases de datos NoSQL) había un ejercicio de una Biblioteca, con base en lo visto en esta guía programe un proyecto nuevo con las clases y sus atributos necesarios para modelar dicho problema.
* Leer sobre los repositorios en el contexto de Spring Boot y sobre MongoRepository.
* Para qué se usa @Indexed, @DBRef o @Field en los documentos de MongoDB en Spring Boot.

Leer:

* <https://medium.com/@diego.coder/relaciones-en-mongodb-edf2107a94ad>
* <https://www.baeldung.com/spring-mongodb-dbref-annotation>