Apunte de Clases 14 - Spring Data

Introducción

Sprint Data es uno de los módulos más grandes de spring framework ya que cuenta con muchos sub módulos y cuyo principal objetivo es proveer una serie de herramientas que permitan realizar integraciones muy sencillas con los diferentes entornos de **almacenamiento de datos**.

Spring Data funciona mediante la creación de interfaces que heredan de las interfaces proporcionadas por Spring. Estas interfaces heredan funcionalidades predefinidas para operaciones comunes de acceso a datos conocidas como **CRUD** (**C**reate-**R**ead-**U**pdate-**D**elete) para crear, leer, actualizar o borrar entidades de una base datos. Lo único que se debe hacer como programador es definir estas interfaces y Spring generará **en tiempo de ejecución** una clase que implementa las funcionalidades indicadas por las interfaces.

Para utilizar Spring Data, se necesita agregar la dependencia correspondiente en el archivo pom.xml del proyecto (ó al momento de inicializar el proyecto desde initializr). Luego, en el archivo **application.properties**, se configuran los datos de conexión a la base de datos, como la URL, el usuario y la contraseña.

Funcionalidades y módulos

El objetivo de Spring Data es reducir la cantidad de código repetitivo que normalmente se tendrían que escribir al acceder a una base de datos. Dentro de las funcionalidades que provee este módulo se pueden mencionar:

- crear potentes repositorios y puedo tener un mapeo de objetos personalizados
- tener consultas dinámicas basadas en el nombre los métodos
- clases base que proporcionan propiedades básicas que facilitan el manejo y el trabajo con los repositorios
- soporte para auditorías transparentes: es decir se puede indicar cuál es el campo o atributo en una clase que hace referencia a la fecha de creación y cuál es el atributo o campo que hace referencia a la fecha de la última modificación del registro/entidad
- por otro lado permite de manera sencilla la integración con spring framework o sea con el core de spring framework

Dentro de los principales módulo de Spring Data se pueden resaltar:

- módulo específico para jpa
- módulo específico con el trabajo con jdbc
- un módulo para el trabajo con el LDAP
- un módulo para el trabajo con MongoDB
- un módulo inclusive que me permite exponer los repositorios como servicios Rest (conocido como **Spring Data Rest**)
- un módulo para la conexión con la Apache Casandra y otro módulo para la conexión con Apache Solar

En Spring Data, las interfaces más comunes utilizadas para crear repositorios son las siguientes:

- CrudRepository: Esta es la interfaz más básica y generalizada para la creación de repositorios. Proporciona métodos para realizar operaciones CRUD (Crear, Leer, Actualizar y Eliminar) en entidades.
- PagingAndSortingRepository: Extiende la interfaz CrudRepository y agrega métodos para la paginación y la clasificación de resultados.
- JpaRepository: Extiende PagingAndSortingRepository y proporciona métodos adicionales para gestionar entidades, como la eliminación por lotes y la gestión de la caché de entidades.
- MongoRepository: Utilizado específicamente para trabajar con bases de datos MongoDB. Ofrece operaciones especializadas para trabajar con documentos MongoDB.
- ElasticsearchRepository: Utilizado para interactuar con el motor de búsqueda Elasticsearch. Proporciona métodos para indexar, buscar y eliminar documentos en Elasticsearch.
- Neo4jRepository: Utilizado para trabajar con bases de datos de grafo Neo4j. Ofrece operaciones para interactuar con nodos y relaciones en la base de datos de grafo.
- CassandraRepository: Utilizado para interactuar con la base de datos Cassandra. Proporciona métodos para realizar operaciones CRUD en tablas de Cassandra.
- RedisRepository: Utilizado para trabajar con bases de datos en memoria como Redis. Ofrece métodos para almacenar y recuperar datos de manera eficiente.

Estas interfaces proporcionan una abstracción sobre las operaciones de almacenamiento y recuperación de datos en la base de datos, lo que permite a los desarrolladores realizar operaciones comunes sin necesidad de escribir consultas SQL o realizar tareas repetitivas de acceso a la base de datos. Cada interfaz se adapta a un tipo específico de base de datos, lo que facilita la integración con la tecnología de almacenamiento de datos que estés utilizando en tu aplicación.

Interfaz CrudRepository

La interfaz `CrudRepository` en Spring Data proporciona una serie de métodos CRUD (Create, Read, Update, Delete) comunes para operaciones de gestión de datos en la base de datos. Algunos de los métodos incluidos en la interfaz `CrudRepository` son:

1. `save(S entity)`: Guarda una entidad en la base de datos. Si la entidad ya existe, se actualiza; de lo contrario, se crea una nueva entrada.
2. `saveAll(Iterable<S> entities)`: Guarda una colección de entidades en la base de datos.
3. `findById(ID id)`: Recupera una entidad por su identificador (clave primaria).
4. `existsById(ID id)`: Verifica si existe una entidad con el identificador proporcionado.
5. `findAll()`: Recupera todas las entidades del tipo especificado.
6. `findAllById(Iterable<ID> ids)`: Recupera todas las entidades cuyos identificadores coinciden con los proporcionados en la lista.
7. `count()`: Obtiene el número total de entidades en la base de datos.

8. `deleteById(ID id)`: Elimina una entidad por su identificador.
9. `delete(T entity)`: Elimina una entidad de la base de datos.
10. `deleteAll()`: Elimina todas las entidades del tipo especificado en la base de datos.
11. `deleteAll(Iterable<? extends T> entities)`: Elimina una colección de entidades de la base de datos.

Estos métodos proporcionan una funcionalidad básica y común para realizar operaciones de lectura, escritura y eliminación en la base de datos. La interfaz `CrudRepository` es una de las interfaces fundamentales de Spring Data y se utiliza como base para crear repositorios personalizados que gestionan entidades específicas en la base de datos.

> Nota: la interfaz JpaRepository agrega un método que en ocasiones suele ser útil, **saveAndFlush()**.Este método, además de guardar la entidad en el contexto de persistencia, fuerza la escritura inmediata de los cambios en la base de datos. En otras palabras, realiza una operación de flush inmediatamente después de guardar la entidad. Esto significa que los cambios se escriben en la base de datos de inmediato y cualquier error de persistencia se detecta en ese momento.

Primer repositorio

Continuando con el caso de estudio citado anteriormente, se propone refactorizar la abstracción de datos inicial utilizando un repositorio de Spring Data JPA. En este caso los datos de las personas registradas al evento serán persistidas en una base de datos MySQL con la siguiente estructura:

- id (long) [PK]
- documento (entero) [no nulo],
- nombre (varchar(50)) [no nulo],
- apellido (varchar(50)) [no nulo],
- fecha de nacimiento (datetime)
- extranjero (char) [no nulo, valor por defecto 'N']

El script de creación del esquema **personas_db** junto con la tabla **personas** se encuentra en el archivo personas.sql

Lo primero será crear un nuevo proyecto de Spring Boot mediante Spring Initializr. Tener presente incluir las dependencias de "Spring Web", "Lombok" y "Spring Data JPA".

Luego se crea una clase Java para representar la entidad **Persona** tal como se muestra en el siguiente código:

```
package com.example.personasData.models;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import java.time.LocalDateTime;
import lombok.Data;

@Entity
```

```java
@Data
public class Persona {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private Long documento;
    private String nombre;
    private String apellido;
    private LocalDateTime fechaNacimiento;
    private char extranjero;
}
```

> *Notas*:
>
> - tener en cuenta que si no se indica lo contrario el atributo fechaNacimiento será mapeado en la tabla como **fecha_nacimiento**.
> - Formato de fecha por defecto para MySql es 'yyyy-mm-dd'

La anotación **@Data** de Lombok genera automáticamente los métodos equals, hashCode, toString, así como los getters y setters para todos los campos de la clase. Esto reduce significativamente la cantidad de código que necesitas escribir manualmente y hace que tu clase sea más concisa y fácil de mantener.

La anotación **@Entity** se utiliza para marcar una clase de Java como una entidad persistente, lo que significa que los objetos de esta clase se guardarán y se recuperarán en una base de datos. Una entidad se mapea generalmente a una tabla en una base de datos relacional. La anotación @Entity debe colocarse encima de una clase que represente un objeto que deseas almacenar en la base de datos.

La anotación **@Id** se utiliza para marcar una propiedad en una clase como la clave primaria de la entidad. La clave primaria es un campo único que identifica de manera única una fila en una tabla de la base de datos. Cada entidad debe tener una propiedad marcada con @Id. Combinada con esta anotación se utiliza **@GeneratedValue** para indicar cómo se generará automáticamente el valor de la clave primaria cuando se inserte una nueva entidad en la base de datos. Puede tomar diferentes estrategias de generación, como GenerationType.IDENTITY, GenerationType.SEQUENCE, GenerationType.AUTO, entre otras, dependiendo del sistema de gestión de bases de datos que estés utilizando.

En la capa de acceso a datos creamos un **repositorio** CRUD para la entidad mediante el siguiente código:

```java
import org.springframework.data.repository.CrudRepository;

public interface PersonaRepository extends CrudRepository<Persona, Long> {
    // Spring Data automáticamente proporciona métodos CRUD básicos
}
```

> Notar que solo se necesita extender de la interfaz CrudRepository indicando la clase y el tipo de datos del campo que será mapeado como clave primaria (CrudRepository<Persona, Long>), sin necesidad de escribir ningún código SQL para las operaciones básicas de manipulación de la entidad.

Por último se configura el conector Java-MySql como una dependencia más del proyecto agregando en el pom.xml del proyecto:

```
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <scope>runtime</scope>
</dependency>
```

A su vez es necesario definir los parámetros de conexión hacia el origen de datos en el archivo application.properties, tal como se muestra en el siguiente código: `

```
spring.datasource.url=jdbc:mysql://localhost:3333/personas_db
spring.datasource.username=111mil
spring.datasource.password=111mil
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

donde:

- **spring.datasource.url**: Esta propiedad define la URL de conexión a la base de datos. En este caso, estás utilizando MySQL como base de datos y conectándote a **localhost** en el puerto **3333** a una base de datos llamada **personas_db**.
- **spring.datasource.username**: Aquí especificas el nombre de usuario utilizado para autenticarte en la base de datos.
- **spring.datasource.password**: Esta propiedad contiene la contraseña asociada al nombre de usuario para la autenticación en la base de datos. Siempre tener en cuenta que almacenar contraseñas en texto plano en archivos de configuración no es una buena práctica de seguridad. En entornos de producción, se recomienda utilizar soluciones de gestión de secrets para manejar las contraseñas de manera segura.
- **spring.datasource.driver-class-name**: Indica la clase del controlador de la base de datos que se utilizará para establecer la conexión. En este caso, estás utilizando el controlador com.mysql.cj.jdbc.Driver.

El código completo del proyecto se encuentra en Spring boot - Personas Data.

> Notas:
>
> - Para probar la ejecución de los servicios expuestos en la API, por defecto se publica en **localhost:8080/api/personas**. Si se quiere cambiar el puerto es necesario agregar el archivo .properties del proyecto la siguiente línea:
>
> ```
> server.port=xxxx
> ```
>
> - Para probar las funcionalidades de la API Personas se deja una colección de pruebas POSTMAN en el siguiente link.

> - Si al intentar registrar una persona se obtiene un código de estado 400 (BAD_REQUEST) posiblemente el campo fechaNacimiento se este mandando en un formato incorrecto.

Consultas dinámicas

Spring Data provee una herramienta en la cual se construyen consultas dinámicas únicamente utilizando un nombramiento de los métodos de un repositorio. Cuando se definen métodos en los repositorios con nombres que comienzan con findBy, Spring Data JPA analiza el nombre del método y genera automáticamente la consulta SQL correspondiente en función del nombre y los parámetros del método. Éstas consultas también suelen ser nombradas como **consultas derivas**.

La estructura general de un método que utiliza findBy es la siguiente:

```
List<Entity> findByPropertyName(ParameterType parameter);
```

Donde:

- **Entity**: Es la entidad JPA sobre la que deseas hacer la consulta.
- **PropertyName**: Es el nombre de un atributo o campo de la entidad por el cual deseas filtrar.
- **ParameterType**: Es el tipo de dato del parámetro que pasas al método para establecer el criterio de búsqueda.

Por ejemplo, continuando con el ejercicio anterior, si se necesita recuperar todas las personas por nombre es posible agregar el siguiente método en la definición de la interfaz de repositorio:

```
 List<Persona> findByNombre(String nombre);
En este caso, el método **findByNombre** generará automáticamente una consulta SQL
que busca todas las personas cuyo nombre coincida con el valor proporcionado.

Si se desea buscar personas por nombre y apellido, es posible combinar los
atributos en el nombre del método:

```Java
public interface PersonaRepository extends JpaRepository<Persona, Long> {
    List<Persona> findByNombreAndApellido(String nombre, String apellido);
}
```

> Nota: cabe resaltar el conector **And** entre los nombres de atributos. Si por ejemlo el método se nombrara findByNombreOrApellido, entonces la búsqueda sería por nombre o apellido coincidentes con los parámetros recibidos.

Para más detalle de palabras clave de repositorio consultar aquí

Además de findBy, Spring Data JPA también admite otros prefijos, como **countBy**, **deleteBy**, entre otros. Estos prefijos permiten realizar operaciones comunes como contar o eliminar registros según ciertos criterios sin necesidad de escribir consultas SQL completas.
```

Es importante tener en cuenta que si se necesita realizar consultas más complejas que no pueden ser generadas automáticamente por los métodos findBy, Spring Data JPA también permite definir consultas personalizadas utilizando la anotación @Query. Esto es más flexibilidad para expresar consultas específicas en lenguaje SQL o JPQL (Java Persistence Query Language).

Consultas mediante @Query

La anotación **@Query** en Spring Data JPA se utiliza para declarar consultas personalizadas que se ejecutarán en la base de datos. Esta anotación permite escribir consultas en lenguaje de consulta específico (como JPQL o SQL nativo) en lugar de depender únicamente de los métodos generados automáticamente por Spring Data JPA. SI las consultas están escritas en SQL puro suelen nombrarse como **consultas nativas**.

En esencia, **@Query** permite definir una consulta personalizada que será utilizada por Spring Data JPA para acceder a los datos en la base de datos. Esto puede ser útil cuando las consultas requeridas son más complejas o no pueden expresarse fácilmente utilizando los métodos de consulta generados automáticamente.

La anotación @Query se puede aplicar en dos niveles:

- **Nivel** de método en el **repositorio**: Puedes aplicar @Query directamente en un método en tu interfaz de repositorio para indicar una consulta específica para ese método.
- **Nivel de entidad** en un repositorio personalizado: También puedes definir un método abstracto en una interfaz de repositorio y proporcionar la implementación en un repositorio personalizado que utilice la anotación @Query.

Por ejemplo supondiendo que se desea agregar al repositorio de Personas una consulta por nombre, apellido y edad, y se desea buscar personas cuyas edades estén dentro de un rango dado y cuyos nombres contengan cierta cadena:

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import java.util.List;

public interface PersonaRepository extends JpaRepository<Persona, Long> {

    @Query("SELECT p FROM Persona p WHERE p.edad BETWEEN :minEdad AND :maxEdad AND
LOWER(p.nombre) LIKE %:nombre%")
    List<Persona> buscarPorEdadYNombre(int minEdad, int maxEdad, String nombre);
}
```

En este ejemplo, se utiliza la anotación @Query para definir la consulta personalizada. La consulta está escrita en JPQL (Java Persistence Query Language), que es un lenguaje similar a SQL pero específico para JPA. La consulta puede leerse como:

- **SELECT p FROM Persona p**: seleccionar entidades Persona y nombrarlas como **p** para su uso en la consulta.

- **WHERE p.edad BETWEEN :minEdad AND :maxEdad**: se está filtrando las personas cuyas edades están dentro del rango especificado por minEdad y maxEdad.
- **AND LOWER(p.nombre) LIKE %:nombre%**: se está agregando otra condición al filtro para que los nombres coincidan (en minúsculas) con la cadena nombre.

Los parámetros **:minEdad**, **:maxEdad** y **:nombre** son marcadores de posición que se vinculan a los parámetros del método buscarPorEdadYNombre.

Para más detalles sobre JPQL consultar la referencia aquí.

Ejemplo: Entradas Kempes

El siguiente ejemplo corresponde a un servicio desarrollado con Springboot y Spring Data para registrar las entradas nominadas a un evento en el Estadio Kempes. El servicio toma los datos de la persona y genera la entrada con los datos para el evento recibido también como argumento. Este ejemplo será retomado en clases posteriores para abordar temas de enrutamiento dinámico a un punto de entrada cuando se tiene un ecosistema de microservicios.

Tomando el mismo esquema de base de datos del ejemplo anterior, es necesario el siguiente comando DDL para la creación de la tabla entradas:

```
CREATE TABLE IF NOT EXISTS `personas_db`.`entradas` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `apellido_nombre` INT NOT NULL,
  `id_evento` INT NOT NULL,
  `socio` CHAR(1) NOT NULL DEFAULT 'N',
  PRIMARY KEY (`id`))
ENGINE = InnoDB;
```

El proyecto Springboot permite crear una API Rest que expone solo un punto de acceso por método POST y recibe los datos de la entrada a registrar. El siguiente ejemplo muestra el formato JSON de la entrada:

```
{
  "evento": {
    "id_evento": 1,
    "nombre": "Concierto de Verano",
    "fecha": "2023-08-15",
    "hora": "19:30"
  },
  "persona": {
    "nombreCompleto": "Juan Pérez",
    "esSocio": "S"
  }
}
```

El código completo del proyecto se encuentra en Spring Data - Entradas Kemples.