Acceso a datos

ORM – Mapeo objeto-relacional

Spring Data / Spring Data JPA - Repositorios

IES Clara del Rey – Madrid

Spring Data JPA

Proyecto de la familia Spring que simplifica el desarrollo de acceso a datos con JPA (Jakarta Persistence API)

Aunque ambos están relacionados con el acceso a datos en Java, hay diferencias clave entre JPA puro y Spring Data JPA.

JPA es una especificación, que define un estándar para el mapeo ORM.

Spring Data JPA agrega una capa de abstracción sobre JPA estándar, facilitando el desarrollo de aplicaciones basadas en Spring que interactúan con bases de datos relacionales.

Simplifica la configuración, y proporciona clases e interfaces de repositorios predefinidos para mejorar la productividad del desarrollador al ofrecer una integración más estrecha con el ecosistema de Spring.

Spring Data JPA – ¿Qué aporta?

Entre otras cosas, Spring Data JPA aporta a JPA:

- Simplificación del código: proporciona un conjunto de interfaces y clases base que se pueden extender para definir las clases de acceso a datos de forma más breve y concisa.
- Repositorios de datos: Introduce el concepto de repositorios, que proporcionan métodos predefinidos para realizar operaciones comunes de CRUD (Create, Read, Update, Delete) sin tener que escribir código.
- Sistema de consultas por convenciones: consultas derivadas de los nombres de los métodos en los repositorios. Utiliza convenciones para interpretar los nombres de los métodos y generar las consultas JPA correspondientes.

Spring Data JPA – ¿Qué aporta?

- Soporte mejorado para transacciones: integración transparente con el sistema de transacciones, para trabajar con ellas de forma declarativa, a base de anotaciones.
- Paginación y ordenación: incorpora paginación y la clasificación de resultados de consultas, sin necesidad de desarrollo a medida.
- Consultas personalizadas: definición de consultas JPQL o SQL nativo de forma simple, basada en anotaciones.

Spring Data, además de para JPA, tiene soporte para otras tecnologías de almacenamiento de datos, como MongoDB, Cassandra, Redis, entre otros.

Repositorio

En términos de ingeniería de software, el repositorio es un patrón de diseño diseño en el que una clase (el repositorio) encapsula la lógica relacionada con el acceso a datos y proporciona una interfaz más simplificada y consistente para la aplicación.

Repositorio – Características principales

- Abstracción de datos: ocultando los detalles específicos de cómo se almacenan y recuperan los datos.
- Operaciones CRUD estandarizadas: operaciones estándar para la creación, lectura, actualización y eliminación de datos.
- Encapsulación de la lógica de acceso a datos: la lógica para interactuar con la fuente de datos (como una base de datos) está encapsulada dentro del repositorio, evitando que la aplicación dependa directamente de los detalles de implementación de la fuente de datos.
- Consistencia en el acceso a datos: interfaz consistente para que la aplicación interactúe con los datos, independientemente de la fuente de datos subyacente.

Repositorio – Características principales

- Facilita las pruebas unitarias: Permite la sustitución de implementaciones de repositorios durante las pruebas unitarias para aislar la lógica de acceso a datos y facilitar las pruebas.
- Desacopla la lógica de negocio: al encapsular la lógica de acceso a datos, el repositorio ayuda a desacoplar la lógica de negocio de los detalles de la persistencia.
- Posibilita cambios en la implementación: Permite cambiar la implementación subyacente (por ejemplo, cambiar de una base de datos relacional a un servicio web) sin afectar la lógica de la aplicación que utiliza el repositorio.

Repositorios en Spring Data

En Spring Data hay seis interfaces principales (hay más) que permiten la creación de repositorios:

- Repository
- CrudRepository y ListCrudrepository
- PagingAndSortingRepository y ListPagingAndSortingRepository
- JpaRepository

Todas las interfaces pertenecen a Spring Data Commons, salvo JpaReposiroy, que es específica de Spring Data JPA.

Hay una serie de relaciones de herencia entre estas interfaces, que veremos más adelante.

Repository <T, ID>

Superclase principal de la jerarquía.

Es un "marker interface". No añade métodos ni abstractos, ni default.

Es un tipo genérico, con los siguientes parámetros de tipo:

- T: clase, entidad (@Entity) que almacenaremos en el repositorio.
- ID: tipo de datos de la clave primaria de la entidad. Será el tipo de datos del campo anotado con @Id. Hay que tener en cuenta:
  - No puede ser un tipo primitivo. Hay que usar el wrapper.
  - En claves compuestas, usaremos la clase que definimos como @Embeddable para la clave, o la clase que usamos con @IdClass.

CrudRepository <T, ID>

Hereda de Repository<T, ID>, y añade los métodos:

- count
- delete, deleteAll, deleteAllById, deleteById
- existsById
- findAll, findAllById, findById
- save, saveAll

Los métodos que devuelven colecciones devuelven Iterable<T>.

La interfaz "ListCrudRepository", que hereda de esta, sobrescribe los métodos findAll, findAllById y saveAll para que devuelvan List<T> en lugar de Iterable<T>.

PagingAndSortingRepository<T, ID>

Hereda de Repository<T, ID>, y añade el método findAll, con dos sobrecargas:

- Iterable<T> findAll(Sort sort): permite obtener entidades ordenadas. Si no queremos ordenar, debemos usar Sort.unsorted() como parámetro.
- Page<T> findAll(Pageable pageable): permite obtener una página de entidades, de cierto tamaño. Si no queremos que se pagine, queremos todas las entidades, debemos usar Pageable.unpaged().

La interfaz Page<T> es la que permite acceder a los datos paginados.

ListPagedAndSortingRepository, que hereda de esta interfaz, sobrescribe el método findAll para que devuelva List<T> en lugar de Iterable<T>.

JpaRepository<T, ID>

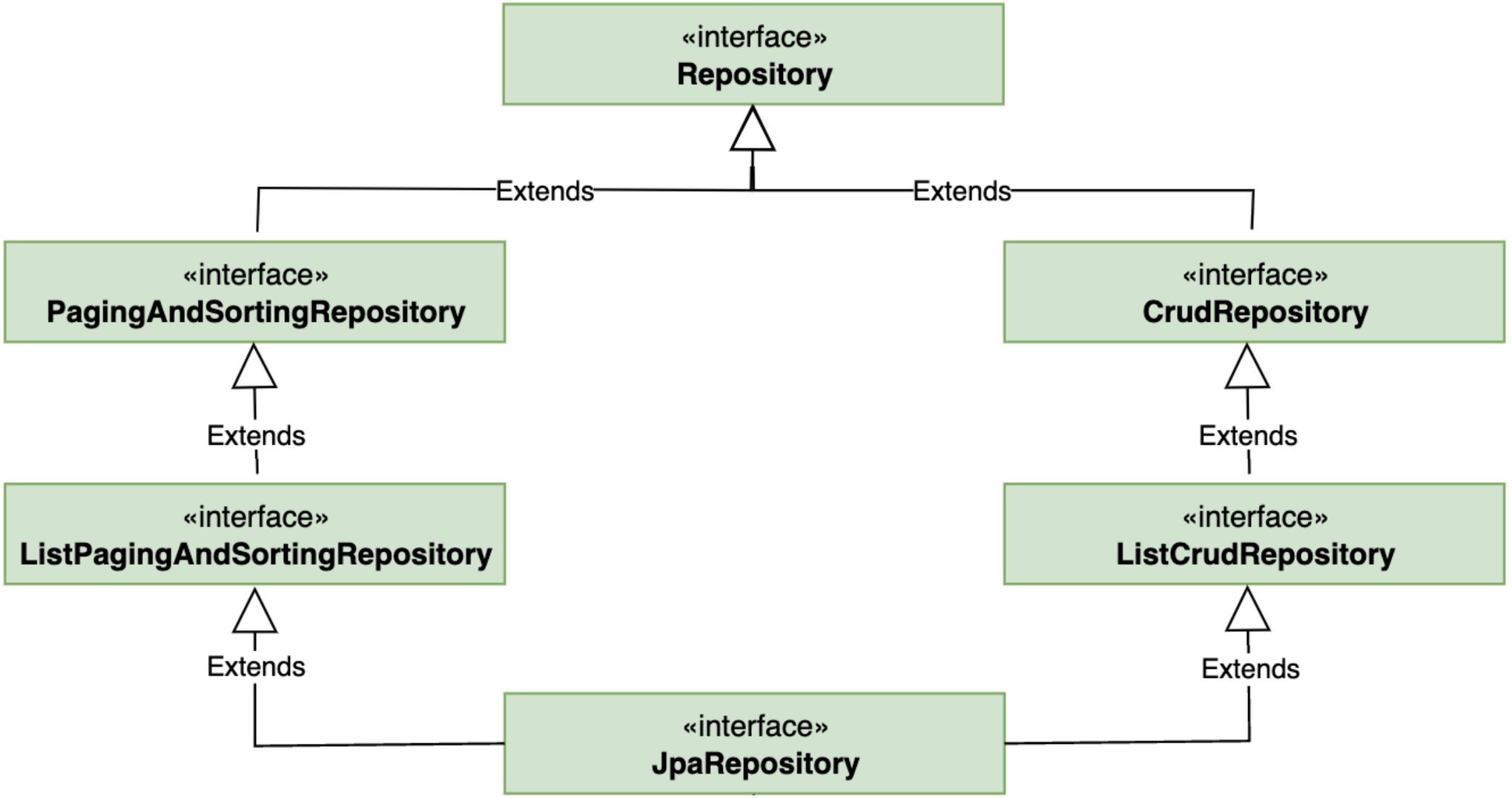
Hereda de:

- ListCrudRepository<T, ID>
- ListPagingAndSortingRepository<T, ID>
- QueryByExampleExecutor<T>

Añade más métodos como (entre otros):

- deleteAllByIdInBatch, deleteAllInBatch, deleteInBatch
- Nuevas sobrecargas de findAll,
- Métodos relacionados con persistencia como flush, saveAllAndFlush, saveAndFlush.

Jerarquía de interfaces



Ejemplo: definición de un repositorio

Supongamos la siguiente clase anotada con @Entity, y el enumerado para colores:

```java
@Entity
public class Coche {
    @Id
    private String matricula;
    private String marca;
    private String modelo;
    @Enumerated(EnumType.STRING)
    private Color color;
}
```

```java
public enum Color {
    ROJO,
    VERDE,
    AZUL,
    AMARILLO,
    OTRO
}
```

Ejemplo: definición de un repositorio

Para crear un repositorio CRUD, hay que crear una interfaz que herede de la interfaz CrudRepository. Ojo, interfaz que hereda, no clase que implementa:

```
public interface CocheRepository extends CrudRepository<Coche, String> {
    // Podemos añadir métodos adicionales
}
```

Y se pueden añadir métodos abstractos que, por ejemplo, operan por convención de nombres (más sobre esto más adelante):

```
public interface CocheRepository extends CrudRepository<Coche, String> {
    // Métodos CRUD generados automáticamente
    // Método de consulta personalizado
    List<Coche> findByMarca(String marca);
}
```