

Llamar SP, formas de usar
queries nativas, criteria queries
y paginación

Llamar SPs

- Hay varias formas de llamar a un SP
- Cuál elegir depende del tipo de Stored
 - Si no tiene parámetros
 - Si tiene sólo parámetros de entrada
 - Sólo de salida
 - Si retorna un resultado (un select)
- Las formas de llamarlo son
 - Con @Procedure
 - Con @Query
 - Con EntityManager
 - Con NamedPareameterJdbcTemplate o con JdbcTemplate

Llamada con @Procedure y con EntityManager

- Tanto @Procedure como EntityManager utilizan namedStoredProcedureQuery para llamar a los SP
- @namedSotredProcedureQuery es una anotación que nos permte predefinir una llamada a un Stored.
- Esta anotación se utiliza en las entidades y nos permite definir
 - Name: nombre que usaremos en la aplicación para llamar al stored
 - procedureName: nombre del stored en la base de datos
 - parameters: define parámetros de entrada
 - resultClasses: nos permite determinar la clase a la que se debe mapear el resultado o con resultSetMappings indicar el mapeo que debe hacerse para los datos devueltos.

Llamada con @Procedure

```
@Entity 7 usages
@Getter
@Setter
@ToString
@NoArgsConstructor
@AllArgsConstructor
@NamedStoredProcedureQuery(
    name = "Person.buscarPersona",
    procedureName = "buscarPersona",
    resultClasses = Person.class,
    parameters = {
        @StoredProcedureParameter(
            mode = ParameterMode.IN,
            name = "person_id",
            type = Long.class)
    }
)
```

```
@Procedure(name = "Person.buscarPersona")
Person buscarPersona(@Param("person_id") Long personId);
```

```
@Procedure 1 usage
Person buscarPersona(@Param("person_id") Long personId);
```

Si el nombre del Stored es igual al del método no hace falta usar @NamedStoredProcedureQuery

Llamada con EntityManager

Edit | Explain | Test | Document | Fix

@AllArgsConstructor 10 usages new *

@Getter

```
public class ResponseStoredDTO {  
  
    private String isbn;  
    private String nombreLibro;  
    private String autor;  
    private LocalDate fechaInicio;  
    private LocalDate fechaFin;  
    private String estado;  
    private String nombreUsuario;  
    private String apellidoUsuario;  
  
}
```

```
@NamedStoredProcedureQuery(  
    name = "listarPrstamosPorEstadoPrestamo",  
    procedureName = "listarPrestamosPorEstado",  
    parameters = {  
        @StoredProcedureParameter(  
            name = "estado",  
            type = String.class,  
            mode = ParameterMode.IN)  
    },  
    resultSetMappings = "getPrestamosPorEstadoPrestamo"  
)  
@SqlResultSetMapping(name = "getPrestamosPorEstadoPrestamo", classes = @ConstructorResult(  
    targetClass = ResponseStoredDTO.class,  
    columns = {  
        @ColumnResult(name = "isbn", type = String.class),  
        @ColumnResult(name = "nombre_libro", type = String.class),  
        @ColumnResult(name = "autor", type = String.class),  
        @ColumnResult(name = "fecha_inicio", type = LocalDate.class),  
        @ColumnResult(name = "fecha_fin", type = LocalDate.class),  
        @ColumnResult(name = "estado", type = String.class),  
        @ColumnResult(name = "nombre_usuario", type = String.class),  
        @ColumnResult(name = "apellido", type = String.class)  
    }  
))
```

Llamada con EntityManager

```
@Repository 3 usages new *
public class PrestamoDao {

    private final EntityManager entityManager; 2 usages

    public PrestamoDao(EntityManager entityManager) { no usages new *
        this.entityManager = entityManager;
    }

    Edit | Explain | Test | Document | Fix
    public List<ResponseStoredDTO> getPrestamosPorEstado(String estado) { 1 usage new *

        return this.entityManager.createNamedQueryQuery("listarPrstamosPorEstadoPrestamo")
            .setParameter(name: "estado", estado)
            .getResultList();
    }
```


Usando JdbcTemplate si solo tiene parámetros

```
public Integer getCantPrestamosDevueltos(){ no usages new *  
  
    Map<String, Object> result = new SimpleJdbcCall(jdbcTemplate)  
        .withProcedureName("cantPrestamosDevueltos")  
        .declareParameters(  
            new SqlParameter(name: "estado", Types.VARCHAR),  
            new SqlOutParameter(name: "cant", Types.INTEGER)  
        )  
        .execute(Map.of(k1: "estado", v1: "devuelto"));  
  
    return (Integer) result.get("cant");  
}
```

Usando JdbcTemplate si retorna datos de una consulta

Edit | Explain | Test | Document | Fix

public class PrestamoRowMapper implements RowMapper<ResponseStoredDTO>{ 2 usages new *

Edit | Explain | Test | Document | Fix

@Override new *

public ResponseStoredDTO mapRow(ResultSet rs, int rowNum) throws SQLException {

return new ResponseStoredDTO(

rs.getString(columnLabel: "isbn"),

rs.getString(columnLabel: "nombre_libro"),

rs.getString(columnLabel: "autor"),

rs.getDate(columnLabel: "fecha_inicio").toLocalDate(),

rs.getDate(columnLabel: "rs_fin").toLocalDate(),

rs.getString(columnLabel: "estado"),

rs.getString(columnLabel: "nombre_usuario"),

rs.getString(columnLabel: "apellido")

);

}

}

Usando JdbcTemplate si retorna datos de una consulta

```
public List<ResponseStoredDTO> getPrestamosPorEstado2(){ no usages new *  
  
    Map<String, Object> result = new SimpleJdbcCall(jdbcTemplate)  
        .withProcedureName("listarPrestamosPorEstado")  
        .declareParameters(  
            new SqlParameter(name: "estado", Types.VARCHAR)  
        )  
        .returningResultSet(parameterName: "prestamosDetalle", new PrestamoRowMapper())  
        .execute(Map.of(k1: "estado", v1: "devuelto"));  
  
    return (List<ResponseStoredDTO>) result.get("prestamosDetalle");  
}
```

Queries nativas

- Más performantes que las queries derivadas o JPQL ya que no requieren el paso de traducción
- Útiles para volúmenes grandes de datos
- Ensucian un poco el código sobre todo si la query tiene subconsultas y varios JOINS
- Se pueden llamar de tres formas distintas
 - Con @Query
 - Con EntityManager
 - Con JdbcTemplate

Queries nativas

- Ejemplos de llamado a queries nativas con Entity Manager

```
Edit | Explain | Test | Document | Fix
public List<Libro> getByAutor(String autor){ no usages new *

    String sql = "SELECT * FROM libro WHERE autor = :autor";

    List<Libro> libros = entityManager.createNativeQuery(sql)
        .setParameter(name: "autor", autor)
        .getResultList();

    return libros;
}
```

Queries nativas

- Ejemplos de llamado a queries nativas con JdbcTemplate

Edit | Explain | Test | Document | Fix

```
public List<LibroDTO> getByNombre(String nombre){ no usages new *

    String sql = "SELECT * FROM libro WHERE nombre = :nombre";


    Map<String, Object> parameters = new HashMap<String, Object>();
    parameters.put("nombre", nombre);

    return namedParameterJdbcTemplate.query(sql, parameters, (rs, rows) ->
        new LibroDTO(
            rs.getLong(columnLabel: "id"),
            rs.getString(columnLabel: "isbn"),
            rs.getString(columnLabel: "nombre"),
            rs.getString(columnLabel: "autor"),
            rs.getInt(columnLabel: "cantidad"),
            rs.getDate(columnLabel: "fecha_creacion").toInstant(),
            rs.getDate(columnLabel: "fecha_modificacion").toInstant()
        )
    );
}
```

Paginación con JPA

- Podemos paginar las consultas tanto si utilizamos patrón DAO como repository
- Spring nos provee la clase Pageable y PageRequest para usarla en los repository
- Si usamos EntityManager tenemos los métodos setFirstResult() y setMaxResults()
- Con JdbcTemplate tenemos que usar Limit y Offset dentro de la query como si la estuvieran ejecutando en la base de datos.

Paginación con JPA

 Edit | Explain | Test | Document | Fix

@Override no usages new *

```
public Page<Libro> findAll(int page, int size) {  
  
    Pageable pageable = PageRequest.of(page, size, Sort.by(...properties: "id").ascending());  
  
    return libroRepository.findAll(pageable);  
}
```


Criteria Query

- Criteria query es una forma librería dentro de Spring Data que nos permite crear queries SEGURAS en tiempo de ejecución.
- Tiene menor performance que las anteriores dado que ahora el framework no solo tiene que traducir la query sino que además debe construirla. Además requiere comprobar los campos opcionales.
- Es muy útil para búsquedas con filtros en los que el usuario puede llenar algunos campos, todos o ninguno.

```
public List<Usuario> obtenerUsuariosOrdenadosPorNombre() {  
    CriteriaBuilder cb = entityManager.getCriteriaBuilder();  
    CriteriaQuery<Usuario> cq = cb.createQuery(Usuario.class);  
    Root<Usuario> root = cq.from(Usuario.class);  
  
    cq.select(root)  
        .orderBy(cb.asc(root.get("nombre")));  
  
    return entityManager.createQuery(cq).getResultList();  
}
```

Select y OrderBy con Criteria Query

```
public List<Usuario> obtenerUsuariosOrdenadosPorNombre() {  
    CriteriaBuilder cb = entityManager.getCriteriaBuilder();  
    CriteriaQuery<Usuario> cq = cb.createQuery(Usuario.class);  
    Root<Usuario> root = cq.from(Usuario.class);  
  
    cq.select(root)  
        .orderBy(cb.asc(root.get("nombre")));  
  
    return entityManager.createQuery(cq).getResultList();  
}
```

Join con Criteria Query

```
import javax.persistence.criteria.JoinType;

public List<Usuario> obtenerUsuariosConLeftJoinPerfil() {
    CriteriaBuilder cb = entityManager.getCriteriaBuilder();
    CriteriaQuery<Usuario> cq = cb.createQuery(Usuario.class);
    Root<Usuario> usuarioRoot = cq.from(Usuario.class);

    // LEFT JOIN con Perfil
    usuarioRoot.join("perfil", JoinType.LEFT);

    // Seleccionar todos los usuarios, incluso si no tienen perfil
    cq.select(usuarioRoot);

    return entityManager.createQuery(cq).getResultList();
}
```

perfil es el nombre del atributo que mapea la relación entre Usuario y Perfil

