Spring Data JPA Repository

jim stafford

Fall 2024 v2022-07-24: Built: 2024-11-19 21:34 EST

Table of Contents

1. Introduction . . . 1
  1.1. Goals . . . 1
  1.2. Objectives . . . 1
2. Spring Data JPA Repository . . . 2
3. Spring Data Repository Interfaces . . . 3
4. SongsRepository . . . 4
  4.1. Song @Entity . . . 4
  4.2. SongsRepository . . . 4
5. Configuration . . . 5
  5.1. Injection . . . 5
6. CrudRepository . . . 6
  6.1. CrudRepository save() New . . . 6
  6.2. CrudRepository save() Update Existing . . . 7
  6.3. CrudRepository save()/Update Resulting SQL . . . 7
  6.4. New Entity? . . . 8
  6.5. CrudRepository existsById() . . . 8
  6.6. CrudRepository findById() . . . 9
  6.7. CrudRepository delete() . . . 10
  6.8. CrudRepository deleteById() . . . 11
  6.9. Other CrudRepository Methods . . . 11
7. PagingAndSortingRepository . . . 13
  7.1. Sorting . . . 13
  7.2. Paging . . . 14
  7.3. Page Result . . . 15
  7.4. Slice Properties . . . 15
  7.5. Page Properties . . . 16
  7.6. Stateful Pageable Creation . . . 16
  7.7. Page Iteration . . . 17
8. Query By Example . . . 18
  8.1. Example Object . . . 18
  8.2. findAll By Example . . . 19
  8.3. Primitive Types are Non-Null . . . 19
  8.4. matchingAny ExampleMatcher . . . 20
  8.5. Ignoring Properties . . . 21
  8.6. Contains ExampleMatcher . . . 21
9. Derived Queries . . . 23
  9.1. Single Field Exact Match Example . . . 23
  9.2. Query Keywords . . . 24

9.3. Other Keywords . . . 24
9.4. Multiple Fields . . . 25
9.5. Collection Response Query Example . . . 25
9.6. Slice Response Query Example . . . 26
9.7. Page Response Query Example . . . 27
10. JPA-QL Named Queries . . . 29
10.1. Mapping @NamedQueries to Repository Methods . . . 29
11. @Query Annotation Queries . . . 31
11.1. @Query Annotation Native Queries . . . 31
11.2. @Query Sort and Paging . . . 32
12. JpaRepository Methods . . . 33
12.1. JpaRepository Type Extensions . . . 33
12.2. JpaRepository flush() . . . 34
12.3. JpaRepository deleteInBatch . . . 35
12.4. JPA References . . . 35
13. Custom Queries . . . 38
13.1. Custom Query Interface . . . 38
13.2. Repository Extends Custom Query Interface . . . 38
13.3. Custom Query Method Implementation . . . 38
13.4. Repository Implementation Postfix . . . 39
13.5. Helper Methods . . . 39
13.6. Naive Injections . . . 40
13.7. Required Injections . . . 40
13.8. Calling Custom Query . . . 41
14. Summary . . . 42
14.1. Comparing Query Types . . . 42

Chapter 1. Introduction

JDBC/SQL provided a lot of capabilities to interface with the database, but with a significant amount of code required. JPA simplified the mapping, but as you observed with the JPA DAO implementation — there was still a modest amount of boilerplate code. Spring Data JPA Repository leverages the capabilities and power of JPA to map `@Entity` classes to the database but also further eliminates much of the boilerplate code remaining with JPA by leveraging Dynamic Interface Proxy techniques.

1.1. Goals

The student will learn:

- to manage objects in the database using the Spring Data Repository
- to leverage different types of built-in repository features
- to extend the repository with custom features when necessary

1.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. declare a `JpaRepository` for an existing JPA `@Entity`
2. perform simple CRUD methods using provided repository methods
3. add paging and sorting to query methods
4. implement queries based on POJO examples and configured matchers
5. implement queries based on predicates derived from repository interface methods
6. implement a custom extension of the repository for complex or compound database access

Chapter 2. Spring Data JPA Repository

Spring Data JPA provides repository support for JPA-based mappings. [1] We start off by writing no mapping code — just interfaces associated with our `@Entity` and primary key type — and have Spring Data JPA implement the desired code. The Spring Data JPA interfaces are layered — offering useful tools for interacting with the database. Our primary `@Entity` types will have a repository interface declared that inherit from `JpaRepository` and any custom interfaces we optionally define.

Figure 1. Spring Data JPA Repository Interfaces

The extends path was modified some with the latest version of Spring Data Commons, but the `JpaRepository` ends up being mostly the same by the time the interfaces get merged at the bottom of the inheritance tree.

[1] *"Spring Data JPA - Reference Documentation"*

Chapter 3. Spring Data Repository Interfaces

As we go through these interfaces and methods, please remember that all of the method implementations of these interfaces (except for custom) will be provided for us.

Repository<T, ID>	marker interface capturing the `@Entity` class and primary key type. Everything extends from this type.
CrudRepository<T,ID>	depicts many of the CRUD capabilities we demonstrated with the JPA DAO in the previous JPA lecture
PagingAndSortingRepository<T, ID>	Spring Data provides some nice end-to-end support for sorting and paging. This interface adds some sorting and paging to the `findAll()` query method provided in `CrudRepository`.
ListPagingAndSortingRepository<T,ID>	overrides the PagingAndSorting-based `Iterable<T>` return type to be a `List<T>`
ListCrudRepository	overrides all CRUD-based `Iterable<T>` return types with `List<T>`
QueryByExampleExecutor<T>	provides query-by-example methods that use prototype `@Entity` instances and configured matchers to locate matching results
JpaRepository<T, ID>	brings together the `CrudRepository`, `PagingAndSortingRepository`, and `QueryByExampleExecutor` interfaces and adds several methods of its own. Unique to JPA, there are methods related to flush and working with JPA references.
SongsRepositoryCustom/ SongsRepositoryCustomImpl	we can write our own extensions for complex or compound calls — while taking advantage of an `EntityManager` and existing repository methods
SongsRepository	our repository inherits from the repository hierarchy and adds additional methods that are automatically implemented by Spring Data JPA

Chapter 4. SongsRepository

All we need to create a functional repository is an `@Entity` class and a primary key type. From our work to date, we know that our `@Entity` is the Song class and the primary key is the primitive `int` type.

4.1. Song @Entity

Song @Entity Example

```
@Entity
@NoArgsConstructor
public class Song {
    @Id //be sure this is jakarta.persistence.Id
    private int id;
```

Use Correct @Id

There are many `@Id` annotation classes. Be sure to be the correct one for the technology you are currently mapping. In this case, use `jakarta.persistence.Id`.

4.2. SongsRepository

We declare our repository at whatever level of `Repository` is appropriate for our use. It would be common to simply declare it as extending `JpaRepository`.

```
public interface SongsRepository extends JpaRepository<Song, Integer> {}① ②
```

① Song is the repository type

② Integer is used for the primary key type for an `int`

Consider Using Non-Primitive Primary Key Types

Although these lecture notes provide ways to mitigate issues with generated primary keys using a primitive data type, you will find that Spring Data JPA works easier with nullable object types.

Repositories and Dynamic Interface Proxies

Having covered the lectures on Dynamic Interface Proxies and have seen the amount of boilerplate code that exists for persistence — you should be able to imagine how the repositories could be implemented with no up-front, compilation knowledge of the `@Entity` type.

Chapter 5. Configuration

As you may have noticed and will soon see, there is a lot triggered by the addition of the repository interface. You should have the state of your source code in a stable state and committed before adding the repository.

Assuming your repository and entity classes are in a package below the class annotated with `@SpringBootApplication` — all that is necessary is the `@EnableJpaRepositories` to enable the necessary auto-configuration to instantiate the repository.

Typical JPA Repository Support Declaration

```
@SpringBootApplication
@EnableJpaRepositories
public class JPASongsApp {
```

If, however, your repository or entities are not located in the default packages scanned, their packages can be scanned with configuration options to the `@EnableJpaRepositories` and `@EntityScan` annotations.

Configuring Repository and @Entity Package Scanning

```
@EnableJpaRepositories(basePackageClasses = {SongsRepository.class}) ① ②
@EntityScan(basePackageClasses = {Song.class}) ① ③
```

① the Java class provided here is used to identify the base Java package

② where to scan for repository interfaces

③ where to scan for `@Entity` classes

5.1. Injection

With the repository interface declared and the JPA repository support enabled, we can then successfully inject the repository into our application.

SongsRepository Injection

```
@Autowired
private SongsRepository songsRepo;
```

Chapter 6. CrudRepository

Let's start looking at the capability of our repository — starting with the declared methods of the `CrudRepository` interface and the return type overrides of the `ListCrudRepository` interface.

CrudRepository<T, ID> and ListCrudRepository<T, ID> Interfaces

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {
  <S extends T> S save(S);
  <S extends T> Iterable<S> saveAll(Iterable<S>);
  Optional<T> findById(ID);
  boolean existsById(ID);
  Iterable<T> findAll();
  Iterable<T> findAllById(Iterable<ID>);
  long count();
  void deleteById(ID);
  void delete(T);
  void deleteAllById(Iterable<? extends ID>);
  void deleteAll(Iterable<? extends T>);
  void deleteAll();
}

public interface ListCrudRepository<T, ID> extends CrudRepository<T, ID> {
  <S extends T> List<S> saveAll(Iterable<S>);
  List<T> findAll();
  List<T> findAllById(Iterable<ID>);
}
```

6.1. CrudRepository save() New

We can use the `CrudRepository.save()` method to either create or update our `@Entity` instance in the database.

In this specific example, we call `save()` with a new object. The JPA provider can tell this is a new object because the generated primary key value is currently unassigned. An object type has a default value of null in Java. Our primitive `int` type has a default value of 0 in Java.

CrudRepository.save() New Example

```
//given an entity instance
Song song = mapper.map(dtoFactory.make());
assertThat(song.getId()).isZero(); ①
//when persisting
songsRepo.save(song);
//then entity is persisted
then(song.getId()).isNotZero(); ②
```

① default value for generated primary key using primitive type interpreted as unassigned

② primary key assigned by provider

The following shows the SQL that is generated by JPA provider to add the new object to the database.

CrudRepository.save() New Example SQL

```
select next value for reposongs_song_sequence
insert into reposongs_song (artist, released, title, id) values (?, ?, ?, ?)
```

6.2. CrudRepository save() Update Existing

The `CrudRepository.save()` method is an "upsert".

- if the `@Entity` is new, the repository will call `EntityManager.persist` as you saw in the previous example
- if the `@Entity` exists, the repository will call `EntityManager.merge` to update the database

CrudRepository.save() Update Existing Example

```
//given an entity instance
Song song = mapper.map(dtoFactory.make());
songsRepo.save(song);
songsRepo.flush(); //for demo only ①
Song updatedSong = Song.builder()
        .id(song.getId()) ③
        .title("new title")
        .artist(song.getArtist())
        .released(song.getReleased())
        .build(); ②
//when persisting update
songsRepo.save(updatedSong);
//then entity is persisted
then(songsRepo.findOne(Example.of(updatedSong))).isPresent(); ④
```

① making sure `@Entity` has been saved

② a new, unmanaged `@Entity` instance is created for a fresh update of the database

③ new, unmanaged `@Entity` instance has an assigned, non-default primary key value

④ object's new state is found in the database

6.3. CrudRepository save()/Update Resulting SQL

The following snippet shows the SQL executed by the repository/EntityManager during the `save()` — where it must first determine if the object exists in the database before calling SQL `INSERT` or `UPDATE`.

CrudRepository.save() Update Existing Example SQL

```
select ... ①
from reposongs_song song0_
where song0_.id=?
binding parameter [1] as [INTEGER] - [1]

update reposongs_song set artist=?, released=?, title=? where id=? ②
binding parameter [1] as [VARCHAR] - [The Beach Boys]
binding parameter [2] as [DATE] - [2010-06-07]
binding parameter [3] as [VARCHAR] - [new title]
binding parameter [4] as [INTEGER] - [1]
```

① `EntityManager.merge()` performs `SELECT` to determine if assigned primary key exists and loads that state

② `EntityManager.merge()` performs `UPDATE` to modify state of existing `@Entity` in database

6.4. New Entity?

We just saw where the same method (`save()`) was used to both create or update the object in the database. This works differently depending on how the repository can determine whether the `@Entity` instance passed to it is new or not.

- for auto-assigned primary keys, the `@Entity` instance is considered new if `@Version` (not used in our example) and `@Id` are not assigned — as long as the `@Id` type is non-primitive.
- for manually assigned and primitive `@Id` types, `@Entity` can implement the `Persistable<ID>` interface to assist the repository in knowing when the `@Entity` is new.

Persistable<ID> Interface

```
public interface Persistable<ID> {
    @Nullable
    ID getId();
    boolean isNew();
}
```

6.5. CrudRepository existsById()

Spring Data JPA adds a convenience method that can check whether the `@Entity` exists in the database without loading the entire object or writing a custom query.

The following snippet demonstrates how we can check for the existence of a given ID.

CrudRepository existsById()

```
//given a persisted entity instance
Song pojoSong = mapper.map(dtoFactory.make());
songsRepo.save(pojoSong);
```

```
//when - determining if entity exists
boolean exists = songsRepo.existsById(pojoSong.getId());
//then
then(exists).isTrue();
```

The following shows the SQL produced from the `findById()` call.

CrudRepository existsById() SQL

```
select count(*) from reposongs_song s1_0 where s1_0.id=? ①
```

① `count(*)` avoids having to return all column values

6.6. CrudRepository findById()

If we need the full object, we can always invoke the `findById()` method, which should be a thin wrapper above `EntityManager.find()`, except that the return type is a Java `Optional<T>` versus the `@Entity` type (`T`).

CrudRepository.findById()

```
//when - finding the existing entity
Optional<Song> result = songsRepo.findById(pojoSong.getId());
//then
then(result).isPresent(); ①
```

① `findById()` always returns a non-null `Optional<T>` object

6.6.1. CrudRepository findById() Found Example

The `Optional<T>` can be safely tested for existence using `isPresent()`. If `isPresent()` returns `true`, then `get()` can be called to obtain the targeted `@Entity`.

Present Optional Example

```
//given
then(result).isPresent();
//when - obtaining the instance
Song dbSong = result.get();
//then - instance provided
then(dbSong).isNotNull();
```

6.6.2. CrudRepository findById() Not Found Example

If `isPresent()` returns `false`, then `get()` will throw a `NoSuchElementException` if called. This gives your code some flexibility for how you wish to handle a target `@Entity` not being found.

Missing Optional Example

```
//given
then(result).isNotPresent();
//then - the optional is asserted during the get()
assertThatThrownBy(() -> result.get())
        .isInstanceOf(NoSuchElementException.class);
```

6.7. CrudRepository delete()

The repository also offers a wrapper around `EntityManager.delete()` where an instance is required. Whether the instance existed or not, a successful call will always result in the `@Entity` no longer in the database.

CrudRepository delete() Example

```
//when - deleting an existing instance
songsRepo.delete(existingSong);
//then - instance will be removed from DB
then(songsRepo.existsById(existingSong.getId())).isFalse();
```

6.7.1. CrudRepository delete() Not Loaded

However, if the instance passed to the `delete()` method is not in its current Persistence Context, then it will load it before deleting so that it has all information required to implement any JPA delete cascade events.

CrudRepository delete() Exists Example SQL

```
select ... from reposongs_song s1_0 where s1_0.id=? ①
delete from reposongs_song where id=?
```

① `@Entity` loaded as part of implementing a delete

JPA Supports Cascade Actions

JPA relationships can be configured to perform an action (e.g., delete) to both sides of the relationship when one side is acted upon (e.g., deleted). This could allow a parent `Album` to be persisted, updated, or deleted with all of its child `Songs` with a single call to the repository/`EntityManager`.

6.7.2. CrudRepository delete() Not Exist

If the instance did not exist, the `delete()` call silently returns.

CrudRepository delete() Does Not Exists Example

```
//when - deleting a non-existing instance
```

```
songsRepo.delete(doesNotExist); ①
```

① no exception thrown for not exist

CrudRepository delete() Does Not Exists Example SQL

```
select ... from reposongs_song s1_0 where s1_0.id=? ①
```

① no `@Entity` was found/loaded as a result of this call

6.8. CrudRepository deleteById()

Spring Data JPA also offers a convenience `deleteById()` method taking only the primary key.

CrudRepository deleteById() Example

```
//when - deleting an existing instance
songsRepo.deleteById(existingSong.getId());
```

However, since this is JPA under the hood and JPA may have cascade actions defined, the `@Entity` is still retrieved if it is not currently loaded in the Persistence Context.

CrudRepository deleteById() Example SQL

```
select ... from reposongs_song s1_0 where s1_0.id=?
delete from reposongs_song where id=?
```

deleteById will Throw Exception

Calling `deleteById` for a non-existant `@Entity` will

- throw a `EmptyResultDataAccessException` <= Spring Boot 3.0.6
- quitely return >= Spring Boot 3.1.0

6.9. Other CrudRepository Methods

That was a quick tour of the `CrudRepository<T,ID>` interface methods. The following snippet shows the methods not covered. Most additional `CrudRepository` methods provide convenience methods around the entire repository. The `ListCrudRepository` override the `Iterable<T>` return type with `List<T>`.

Other CrudRepository Methods

```
//public interface CrudRepository<T, ID> extends Repository<T, ID> {
<S extends T> Iterable<S> saveAll(Iterable<S>);
Iterable<T> findAll();
Iterable<T> findAllById(Iterable<ID>);
long count();
```

```java
void deleteAll(Iterable<? extends T>);
void deleteAll();

//public interface ListCrudRepository<T, ID> extends CrudRepository<T, ID> {
<S extends T> List<S> saveAll(Iterable<S>);
List<T> findAll();
List<T> findAllById(Iterable<ID>);
```

Chapter 7. PagingAndSortingRepository

Before we get too deep into queries, it is good to know that Spring Data has first-class support for sorting and paging.

- **sorting** - determines the order which matching results are returned
- **paging** - breaks up results into chunks that are easier to handle than entire database collections

Here is a look at the declared methods of the `PagingAndSortingRepository<T,ID>` interface and the `ListPagingAndSortingRepository<T,ID>` overrides. These interfaces define findAll() methods that accept paging and sorting parameters and return types. They define extra parameters not included in the `CrudRepository` findAll() methods.

PagingAndSortingRepository<T,ID> Interface

```
public interface PagingAndSortingRepository<T, ID> extends Repository<T, ID> {
  Iterable<T> findAll(Sort);
  Page<T> findAll(Pageable);
}

public interface ListPagingAndSortingRepository<T, ID> extends
PagingAndSortingRepository<T, ID> {
  List<T> findAll(Sort);
}
```

We will see the paging and sorting option come up in many other query types as well.

Use Paging and Sorting for Collection Queries

All queries that return a collection should seriously consider adding paging and sorting parameters. Small test databases can become significantly populated production databases over time and cause eventual failure if paging and sorting are not applied to unbounded collection query return methods.

7.1. Sorting

Sorting can be performed on one or more properties and in ascending and descending order.

The following snippet shows an example of calling the `findAll()` method and having it return

- `Song` entities in descending order according to `release` date
- `Song` entities in ascending order according to `id` value when `release` dates are equal

Sort.by() Example

```
//when
List<Song> byReleased = songsRepository.findAll(
        Sort.by("released").descending().and(Sort.by("id").ascending())); ① ②
//then
```

```
LocalDate previous = null;
for (Song s: byReleased) {
    if (previous!=null) {
        then(previous).isAfterOrEqualTo(s.getReleased()); //DESC order
    }
    previous=s.getReleased();
}
```

① results can be sorted by one or more properties

② order of sorting can be ascending or descending

The following snippet shows how the SQL was impacted by the `Sort.by()` parameter.

Sort.by() Example SQL

```
select ...
from reposongs_song s1_0
order by s1_0.released desc,s1_0.id ①
```

① Sort.by() added the extra SQL `order by` clause

7.2. Paging

Paging permits the caller to designate how many instances are to be returned in a call and the offset to start that group (called a page or slice) of instances.

The snippet below shows an example of using one of the factory methods of `Pageable` to create a `PageRequest` definition using page size (limit), offset, and sorting criteria. If many pages are traversed — it is advised to sort by a property that will produce a stable sort over time during table modifications.

Defining Initial Pageable

```
//given
int offset = 0;
int pageSize = 3;
Pageable pageable = PageRequest.of(offset/pageSize, pageSize, Sort.by("released"));①
②
//when
Page<Song> songPage = songsRepository.findAll(pageable);
```

① using `PageRequest` factory method to create `Pageable` from provided page information

② parameters are pageNumber, pageSize, and Sort

Use Stable Sort over Large Collections

Try to use a property for sort (at least by default) that will produce a stable sort when paging through a large collection to avoid repeated or missing objects from follow-on pages because of new changes to the table.

7.3. Page Result

The page result is represented by a container object of type `Page<T>`, which extends `Slice<T>`. I will describe the difference next, but the `PagingAndSortingRepository<T,ID>` interface always returns a `Page<T>`, which will provide:

- the sequential number of the page/slice
- the requested size of the page/slice
- the number of elements found
- the total number of elements available in the database

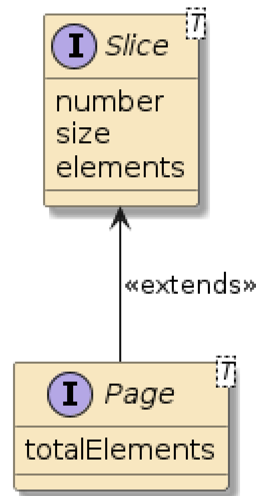

Figure 2. Page<T> Extends Slice<T>

Page Issues Extra Count Query

Of course, the total number of elements available in the database does not come for free. An extra query is performed to get the count. If that attribute is unnecessary, use a Slice return using a derived query.

7.4. Slice Properties

The `Slice<T>` base interface represents properties about the content returned.

Slice Properties

```
//then
Slice songSlice = songPage; ①
then(songSlice).isNotNull();
then(songSlice.isEmpty()).isFalse();
then(songSlice.getNumber()).isEqualTo(0); ②
then(songSlice.getSize()).isEqualTo(pageSize); ③
then(songSlice.getNumberOfElements()).isEqualTo(pageSize); ④

List<Song> songsList = songSlice.getContent();
then(songsList).hasSize(pageSize);
```

① `Page<T>` extends `Slice<T>`

② slice increment — first slice is 0

③ the number of elements requested for this slice

④ the number of elements returned in this slice

7.5. Page Properties

The Page<T> derived interface represents properties about the entire collection/table.

The snippet below shows an example of the total number of elements in the table being made available to the caller.

Page Properties

```
then(songPage.getTotalElements()).isEqualTo(savedSongs.size()); //unique to Page
```

The Page<T> content and number of elements is made available through the following set of SQL queries.

Page Resulting SQL

```
select ... from reposongs_song s1_0 ①
  order by s1_0.released
  offset ? rows fetch first ? rows only

select count(s1_0.id) from reposongs_song s1_0 ②
```

① SELECT used to load page of entities (aka the Slice information)

② SELECT COUNT(*) used to return total matches in the database — returned or not because of Pageable limits (aka the Page portion of the information)

7.6. Stateful Pageable Creation

In the above example, we created a Pageable from stateless parameters — passing in pageNumber, pageSize, and sorting specifications.

Review: Stateless Pageable Definition

```
Pageable pageable = PageRequest.of(offset / pageSize, pageSize, Sort.by("released"))
;①
```

① parameters are pageNumber, pageSize, and Sort

We can also use the original Pageable to generate the next or other relative page specifications.

Relative Pageable Creation

```
Pageable next = pageable.next();
Pageable previous = pageable.previousOrFirst();
Pageable first = pageable.first();
```

7.7. Page Iteration

The next Pageable can be used to advance through the complete set of query results, using the previous Pageable and testing the returned Slice.

Page Iteration

```
for (int i=1; songSlice.hasNext(); i++) { ①
    pageable = pageable.next(); ②
    songSlice = songsRepository.findAll(pageable);
    songsList = songSlice.getContent();
    then(songSlice).isNotNull();
    then(songSlice.getNumber()).isEqualTo(i);
    then(songSlice.getSize()).isEqualTo(pageSize);
    then(songSlice.getNumberOfElements()).isLessThanOrEqualTo(pageSize);
    then(((Page)songSlice).getTotalElements()).isEqualTo(savedSongs.size());//unique
to Page
}
then(songSlice.hasNext()).isFalse();
then(songSlice.getNumber()).isEqualTo(songsRepository.count() / pageSize);
```

① Slice.hasNext() will indicate when previous Slice represented the end of the results

② next Pageable obtained from previous Pageable

The following snippet shows an example of the SQL issued to the database with each page request.

Page Iteration SQL

```
select ... from reposongs_song s1_0
  order by s1_0.released
  offset ? rows fetch first ? rows only
--binding parameter [1] as [INTEGER] - [6] ①
--binding parameter [2] as [INTEGER] - [3]

select count(s1_0.id) from reposongs_song s1_0
```

① paging advances offset

Chapter 8. Query By Example

Not all queries will be as simple as `findAll()`. We now need to start looking at queries that can return a subset of results based on them matching a set of predicates. The `QueryByExampleExecutor<T>` parent interface to `JpaRepository<T,ID>` provides a set of variants to the collection-based results that accepts an "example" to base a set of predicates off of.

QueryByExampleExecutor<T> Interface

```
public interface QueryByExampleExecutor<T> {
  <S extends T> Optional<S> findOne(Example<S>);
  <S extends T> Iterable<S> findAll(Example<S>);
  <S extends T> Iterable<S> findAll(Example<S>, Sort);
  <S extends T> Page<S> findAll(Example<S>, Pageable);
  <S extends T> long count(Example<S>);
  <S extends T> boolean exists(Example<S>);
  <S extends T, R> R findBy(Example<S>, Function<FluentQuery$FetchableFluentQuery<S>,
R>);
}
```

8.1. Example Object

An `Example` is an interface with the ability to hold onto a probe and matcher.

8.1.1. Probe Object

The probe is an instance of the repository `@Entity` type.

The following snippet is an example of creating a probe that represents the fields we are looking to match.

Probe Example

```
//given
Song savedSong = savedSongs.get(0);
Song probe = Song.builder()
        .title(savedSong.getTitle())
        .artist(savedSong.getArtist())
        .build(); ①
```

① probe will carry values for `title` and `artist` to match

8.1.2. ExampleMatcher Object

The matcher defaults to an exact match of all non-null properties in the probe. There are many definitions we can supply to customize the matcher.

- `ExampleMatcher.matchingAny()` - forms an OR relationship between all predicates

* `ExampleMatcher.matchingAll()` - forms an AND relationship between all predicates

The matcher can be broken down into specific fields, designing a fair number of options for String-based predicates but very limited options for non-String fields.

* exact match
* case-insensitive match
* starts with, ends with
* contains
* regular expression
* include or ignore nulls

The following snippet shows an example of the default `ExampleMatcher`.

Default ExampleMatcher

```
ExampleMatcher matcher = ExampleMatcher.matching(); ①
```

① default matcher is `matchingAll`

8.2. findAll By Example

We can supply an `Example` instance to the `findAll()` method to conduct our query.

The following snippet shows an example of using a probe with a default matcher. It is intended to locate all songs matching the `artist` and `title` we specified in the probe.

```
//when
List<Song> foundSongs = songsRepository.findAll(
        Example.of(probe),//default matcher is matchingAll() and non-null
        Sort.by("id"));
```

However, there is a problem. Our `Example` instance with supplied probe and default matcher did not locate any matches.

No Matches Found - huh?

```
//then - not found
then(foundSongs).isEmpty();
```

8.3. Primitive Types are Non-Null

The reason for the no-match is that the primary key value is being added to the query, and we did not explicitly supply that value in our probe.

No Matches SQL

```
select ...  from reposongs_song s1_0
  where s1_0.id=? --filled in with 0 ①
  and s1_0.artist=? and s1_0.title=?
```

```
  order by s1_0.id
--binding parameter [1] as [INTEGER] - [0]
--binding parameter [2] as [VARCHAR] - [Creedence Clearwater Revival]
--binding parameter [3] as [VARCHAR] - [Quo Vadis green]
```

① `id=0` test for unassigned primary key, prevents match being found

The `id` field is a primitive `int` type that cannot be null and defaults to a 0 value. That, and the fact that the default matcher is a "match all" (using `AND`) keeps our example from matching anything.

@Entity Uses Primitive Type for Primary Key

```
@Entity
public class Song {
    @Id @GeneratedValue
    private int id; ①
```

① `id` can never be null and defaults to 0, unassigned value

8.4. matchingAny ExampleMatcher

One option we could take would be to switch from the default `matchingAll` matcher to a `matchingAny` matcher.

The following snippet shows an example of how we can specify the override.

matchingAny ExampleMatcher Example

```
//when
List<Song> foundSongs = songsRepository.findAll(
        Example.of(probe, ExampleMatcher.matchingAny()),①
        Sort.by("id"));
```

① using `matchingAny` versus default `matchingAll`

This causes some matches to occur, but it likely is not what we want.

- the `id` predicate is still being supplied
- the overall condition does not require the `artist` **AND** `title` to match.

matchingAny ExampleMatcher Example SQL

```
select ...
from reposongs_song s1_0
where s1_0.id=? --filled in with 0 ①
  or s1_0.artist=? or s1_0.title=?
order by s1_0.id
```

① matching any (**"or"**) of the non-null probe values

8.5. Ignoring Properties

What we want to do is use a `matchAll` matcher and have the non-null primitive `id` field ignored.

The following snippet shows an example matcher configured to ignore the primary key.

matchingAll ExampleMatcher with Ignored Property

```
ExampleMatcher ignoreId = ExampleMatcher.matchingAll().withIgnorePaths("id");①
//when
List<Song> foundSongs = songsRepository.findAll(
        Example.of(probe, ignoreId), ②
        Sort.by("id"));
//then
then(foundSongs).isNotEmpty();
then(foundSongs.get(0).getId()).isEqualTo(savedSong.getId());
```

① `id` primary key is being excluded from predicates

② non-null and non-id fields of probe are used for **AND** matching

The following snippet shows the SQL produced. This SQL matches only the `title` and `artist` fields, without a reference to the `id` field.

matchingAll ExampleMatcher with Ignored Property SQL

```
select ...
from reposongs_song s1_0
where s1_0.artist=? and s1_0.title=? ① ②
order by s1_0.id
```

① the primitive `int id` field is being ignored

② both `title` and `artist` fields must match

8.6. Contains ExampleMatcher

We have some options on what we can do with the String matches.

The following snippet provides an example of testing whether `title` contains the text in the probe while performing an exact match of the `artist` and ignoring the `id` field.

Contains ExampleMatcher

```
Song probe = Song.builder()
        .title(savedSong.getTitle().substring(2))
        .artist(savedSong.getArtist())
        .build();
ExampleMatcher matcher = ExampleMatcher
        .matching()
        .withIgnorePaths("id")
```

```
        .withMatcher("title", ExampleMatcher.GenericPropertyMatchers.contains());
```

8.6.1. Using Contains ExampleMatcher

The following snippet shows that the `Example` successfully matched on the `Song` we were interested in.

Example is Found

```
//when
List<Song> foundSongs = songsRepository.findAll(Example.of(probe,matcher), Sort.by("
id"));
//then
then(foundSongs).isNotEmpty();
then(foundSongs.get(0).getId()).isEqualTo(savedSong.getId());
```

The following SQL shows what was performed by our `Example`. Both `title` and `artist` are required to match. The match for `title` is implemented as a "contains" `LIKE`.

Contains Example SQL

```
select ...
from reposongs_song s1_0
where s1_0.artist=? and s1_0.title like ? escape '\' ②
order by s1_0.id
//binding parameter [1] as [VARCHAR] - [Earth Wind and Fire]
//binding parameter [2] as [VARCHAR] - [% a God Unknown%] ①
```

① title parameter supplied with `%` characters around the probe value

② title predicate uses a `LIKE`

Chapter 9. Derived Queries

For fairly straight forward queries, Spring Data JPA can derive the required commands from a method signature declared in the repository interface. This provides a more self-documenting version of similar queries we could have formed with query-by-example.

The following snippet shows a few example queries added to our repository interface to address specific queries needed in our application.

Example Query Method Names

```
public interface SongsRepository extends JpaRepository<Song, Integer> {
    Optional<Song> getByTitle(String title); ①

    List<Song> findByTitleNullAndReleasedAfter(LocalDate date); ②

    List<Song> findByTitleStartingWith(String string, Sort sort); ③
    Slice<Song> findByTitleStartingWith(String string, Pageable pageable); ④
    Page<Song> findPageByTitleStartingWith(String string, Pageable pageable); ⑤
```

① query by an exact match of `title`

② query by a match of two fields

③ query using sort

④ query with paging support

⑤ query with paging support and table total

Let's look at a complete example first.

9.1. Single Field Exact Match Example

In the following example, we have created a query method `getByTitle` that accepts the exact match title value and an `Optional` return value.

Interface Method Signature

```
Optional<Song> getByTitle(String title); ①
```

We use the declared interface method normally, and Spring Data JPA takes care of the implementation.

Interface Method Usage

```
//when
Optional<Song> result = songsRepository.getByTitle(song.getTitle());
//then
then(result).isPresent();
```

The resulting SQL is the same as if we implemented it using query-by-example or JPA query language.

Resulting SQL

```
select ...
  from reposongs_song s1_0
  where s1_0.title=?
```

9.2. Query Keywords

Spring Data has several keywords, followed by `By`, that it looks for starting the interface method name. Those with multiple terms can be used interchangeably.

Meaning	Keywords
Query	• find • get • search • read • query • stream
Count	• count
Exists	• exists
Delete	• delete • remove

9.3. Other Keywords

Other keywords include [1] [2]

- Distinct (e.g., `findDistinctByTitle`)
- Is, Equals (e.g., `findByTitle`, `findByTitleIs`, `findByTitleEquals`)
- Not (e.g., `findByTitleNot`, `findByTitleIsNot`, `findByTitleNotEquals`)
- IsNull, IsNotNull (e.g., `findByTitle(null)`, `findByTitleIsNull()`, `findByTitleIsNotNull()`)
- StartingWith, EndingWith, Containing (e.g., `findByTitleStartingWith`, `findByTitleEndingWith`, `findByTitleContaining`)
- LessThan, LessThanEqual, GreaterThan, GreaterThanEqual, Between (e.g., `findByIdLessThan`, `findByIdBetween(lo,hi)`)
- Before, After (e.g., `findByReleaseAfter`)
- In (e.g., `findByTitleIn(collection)`)
- OrderBy (e.g., `findByTitleContainingOrderByTitle`)

The list is significant but not meant to be exhaustive. Perform a web search for your specific needs (e.g., "Spring Data Derived Query ...") if what is needed is not found here.

9.4. Multiple Fields

We can define queries using one or more fields using `And` and `Or`.

The following example defines an interface method that will test two fields: `title` and `released`. `title` will be tested for null and `released` must be after a certain date.

Multiple Fields Interface Method Declaration

```
List<Song> findByTitleNullAndReleasedAfter(LocalDate date);
```

The following snippet shows an example of how we can call/use the repository method. We are using a simple collection return without sorting or paging.

Multiple Fields Example Use

```
//when
List<Song> foundSongs = songsRepository.findByTitleNullAndReleasedAfter(firstSong
.getReleased());
//then
Set<Integer> foundIds = foundSongs.stream()
    .map(s->s.getId())
    .collect(Collectors.toSet());
then(foundIds).isEqualTo(expectedIds);
```

The resulting SQL shows that a query is performed looking for null `title` and `released` after the LocalDate provided.

Multiple Fields Resulting SQL

```
select ...
  from reposongs_song s1_0
  where s1_0.title is null and s1_0.released>?
```

9.5. Collection Response Query Example

We can perform queries with various types of additional arguments and return types. The following shows an example of a query that accepts a sorting order and returns a simple collection with all objects found.

Collection Response Interface Method Declaration

```
List<Song> findByTitleStartingWith(String string, Sort sort);
```

The following snippet shows an example of how to form the `Sort` and call the query method derived from our interface declaration.

Collection Response Interface Method Use

```
//when
Sort sort = Sort.by("id").ascending();
List<Song> songs = songsRepository.findByTitleStartingWith(startingWith, sort);
//then
then(songs.size()).isEqualTo(expectedCount);
```

The following shows the resulting SQL — which now contains a sort clause based on our provided definition.

Collection Response Resulting SQL

```
select ...
  from reposongs_song s1_0
  where s1_0.title like ? escape '\'
  order by s1_0.id
```

9.6. Slice Response Query Example

Derived queries can also be declared to accept a `Pageable` definition and return a `Slice`. The following example shows a similar interface method declaration to what we had prior — except we have wrapped the `Sort` within a `Pageable` and requested a `Slice`, which will contain only those items that match the predicate and comply with the paging constraints.

Slice Response Interface Method Declaration

```
Slice<Song> findByTitleStartingWith(String string, Pageable pageable);
```

The following snippet shows an example of forming the `PageRequest`, making the call, and inspecting the returned `Slice`.

Slice Response Interface Method Use

```
//when
PageRequest pageable=PageRequest.of(0, 1, Sort.by("id").ascending());
Slice<Song> songsSlice=songsRepository.findByTitleStartingWith(startingWith, pageable
);
//then
then(songsSlice.getNumberOfElements()).isEqualTo(pageable.getPageSize());
```

The following resulting SQL shows how paging offset and limits were placed in the query.

Slice Response Resulting SQL

```
select ...
  from reposongs_song s1_0
  where s1_0.title like ? escape '\'
```

```
  order by s1_0.id
  offset ? rows fetch first ? rows only
--binding parameter [1] as [VARCHAR] - [F%]
--binding parameter [2] as [INTEGER] - [0]
--binding parameter [3] as [INTEGER] - [2]
```

9.7. Page Response Query Example

We can alternatively declare a `Page` return type if we also need to know information about all available matches in the table. The following shows an example of returning a `Page`. The only reason `Page` shows up in the method name is to form a different method signature than its sibling examples. `Page` is not required to be in the method name.

Page Response Interface Method Declaration

```
Page<Song> findPageByTitleStartingWith(String string, Pageable pageable);
```

The following snippet shows how we can form a `PageRequest` to pass to the derived query method and accept a `Page` in reponse with additional table information.

Page Response Interface Method Use

```
//when
PageRequest pageable = PageRequest.of(0, 1, Sort.by("id").ascending());
Page<Song> songsPage = songsRepository.findPageByTitleStartingWith(startingWith,
pageable);
//then
then(songsPage.getNumberOfElements()).isEqualTo(pageable.getPageSize());
then(songsPage.getTotalElements()).isEqualTo(expectedCount); ①
```

① an extra property is available to tell us the total number of matches relative to the entire table — that may not have been reported on the current page

The following shows the resulting SQL of the `Page` response. Note that two queries were performed. One provided all the data required for the parent `Slice` and the second query provided the table totals not bounded by the page limits.

Page Response Resulting SQL

```
select ... ①
  from reposongs_song s1_0
  where s1_0.title like ? escape '\'
  order by s1_0.id
  offset ? rows fetch first ? rows only
--binding parameter [1] as [VARCHAR] - [T%]
--binding parameter [2] as [INTEGER] - [0]
--binding parameter [3] as [INTEGER] - [1]

select count(s1_0.id) ②
```

```
  from reposongs_song s1_0
  where s1_0.title like ? escape '\'
--binding parameter [1] as [VARCHAR] - [T%]
```

① first query provides `Slice` data within `Pageable` limits (offset ommitted for first page)

② second query provides table-level count for `Page` that have no page size limits

[1] *"Query Creation"*, Spring Data JPA - Reference Documentation

[2] *"Derived Query Methods in Spring Data JPA"*, Atta

Chapter 10. JPA-QL Named Queries

Query-by-example and derived queries are targeted at flexible but mostly simple queries. Often there is a need to write more complex queries. If you remember in JPA, we can write JPA-QL and native SQL queries to implement our database query access. We can also register them as a `@NamedQuery` associated with the `@Entity` class. This allows for more complex queries as well as to use queries defined in a JPA `orm.xml` source file (without having to recompile)

The following snippet shows a `@NamedQuery` called `Song.findArtistGESize` that implements a query of the `Song` entity's table to return `Song` instances that have artist names longer than a particular size.

JPA-QL @NamedQuery Can Express More Complex Queries

```
@Entity
@Table(name="REPOSONGS_SONG")
@NamedQuery(name="Song.findByArtistGESize",
        query="select s from Song s where length(s.artist) >= :length")
public class Song {
```

The following snippet shows an example of using that `@NamedQuery` with the JPA `EntityManager`.

JPA Named Query Syntax

```
TypedQuery<Song> query = entityManager
        .createNamedQuery("Song.findByArtistGESize", Song.class)
        .setParameter("length", minLength);
List<Song> jpaFoundSongs = query.getResultList();
```

10.1. Mapping @NamedQueries to Repository Methods

That same tool is still available to us with repositories. If we name the query `[prefix].[suffix]`, where `prefix` is the `@Entity.name` of the objects returned and `suffix` matches the name of the repository interface method — we can have them automatically called by our repository.

The following snippet shows a repository interface method that will have its query defined by the `@NamedQuery` defined on the `@Entity` class. Note that we map repository method parameters to the `@NamedQuery` parameter using the `@Param` annotation.

Repository Interface Methods can Automatically Invoke Matching @NamedQueries

```
//see @NamedQuery(name="Song.findByArtistGESize" in Song class
List<Song> findByArtistGESize(@Param("length") int length); ① ②
```

① interface method name matches `@NamedQuery.name" suffix

② `@Param` maps method parameter to `@NamedQuery` parameter

The following snippet shows the resulting SQL generated from the JPA-QL/`@NamedQuery`

JPA-QL Resulting SQL

```
select ...
  from reposongs_song s1_0
  where character_length(s1_0.artist)>=?
```

Chapter 11. @Query Annotation Queries

Spring Data JPA provides an option for the query to be expressed on the repository method versus the `@Entity` class.

The following snippet shows an example of a similar query we did for `artist` length — except in this case we are querying against `title` length.

Query Supplied on Repository Method

```
@Query("select s from Song s where length(s.title) >= :length")
List<Song> findByTitleGESize(@Param("length") int length);
```

We get the expected resulting SQL.

Resulting SQL

```
select ...
  from reposongs_song s1_0
  where character_length(s1_0.artist)>=?
```

Named Queries can be supplied in property file

Named queries can also be expressed in a property file — versus being placed directly onto the method. Property files can provide a more convenient source for expressing more complex queries.

```
@EnableJpaRepositories(namedQueriesLocation="...")
```

The default location is `META-INF/jpa-named-queries.properties`

11.1. @Query Annotation Native Queries

Although I did not demonstrate it, the `@NamedQuery` can also be expressed in native SQL. In most cases with native SQL queries, the returned information is just data. We can also directly express the repository interface method as a native SQL query as well as have it returned straight data.

The following snippet shows a repository interface method implemented as native SQL that will return only the `title` columns based on size.

Example Native SQL @Query Method

```
@Query(value="select s.title from REPOSONGS_SONG s where length(s.title) >= :length",
nativeQuery=true)
List<String> getTitlesGESizeNative(@Param("length") int length);
```

The following output shows the resulting SQL. We can tell this was from a native SQL query

because the SQL does not contain mangled names used by JPA generated SQL.

Resulting Native SQL

```
select s.title ①
from REPOSONGS_SONG s
where length(s.title) >= ?
```

① native SQL query gets expressed exactly as we supplied it

11.2. @Query Sort and Paging

The `@Query` approach supports paging via `Pageable` parameter. Sort must be defined within the query.

@Query Sort and Paging

```
@Query(value="select s from Song s where released between :starting and :ending order
by id ASC")
Page<Song> findByReleasedBetween(LocalDate starting, LocalDate ending, Pageable
pageable);
```

Chapter 12. JpaRepository Methods

Many of the methods and capabilities of the `JpaRepository<T,ID>` are available at the higher level interfaces. The `JpaRepository<T,ID>` itself declares four types of additional methods

- flush-based methods
- batch-based deletes
- reference-based accessors
- return type extensions

JpaRepository<T, ID> Interface

```
public interface JpaRepository<T, ID> extends ListCrudRepository<T, ID>,
ListPagingAndSortingRepository<T, ID>, QueryByExampleExecutor<T> {
  void flush();
  <S extends T> S saveAndFlush(S);
  <S extends T> List<S> saveAllAndFlush(Iterable<S>);

  void deleteAllInBatch(Iterable<T> entities);
  void deleteAllByIdInBatch(Iterable<ID>);
  void deleteAllInBatch();

  T getReferenceById(ID);

  <S extends T> List<S> findAll(Example<S>);
  <S extends T> List<S> findAll(Example<S>, Sort);
}
```

12.1. JpaRepository Type Extensions

The methods in the `JpaRepository<T,ID>` interface not discussed here mostly just extend existing parent methods with more concrete return types (e.g., `List` versus `Iterable`).

Abstract Generic Spring Data Methods

```
public interface QueryByExampleExecutor<T> {
    <S extends T> Iterable<S> findAll(Example<S> example);
```

Concrete Spring Data JPA Extensions

```
public interface JpaRepository<T, ID> extends ..., QueryByExampleExecutor<T> {
    @Override
    <S extends T> List<S> findAll(Example<S> example); ①
   ...
```

① `List<T>` extends `Iterable<T>`

12.2. JpaRepository flush()

As we know with JPA, many commands are cached within the local Persistence Context and issued to the database at some point in time in the future. That point in time is either the end of the transaction or some event within the scope of the transaction (e.g., issue a JPA query). `flush()` commands can be used to immediately force queued commands to the database. We would need to do this before issuing a native SQL command if we want our latest changes to be included with that command.

In the following example, a transaction is held open during the entire method because of the `@Transaction` declaration. `saveAll()` just adds the objects to the Persistence Context and caches their insert commands. The `flush()` command finally forces the SQL `INSERT` commands to be issued.

```
@Test
@Transactional
void flush() {
    //given
    List<Song> songs = dtoFactory.listBuilder().songs(5,5).stream()
                .map(s->mapper.map(s))
                .collect(Collectors.toList());
    songsRepository.saveAll(songs); ①
    //when
    songsRepository.flush(); ②
}
```

① instances are added to the Persistence Unit cache

② instances are explicitly flushed to the database

The pre-flush actions are only to assign the primary key value.

Database Calls Pre-Flush

```
Hibernate: select next value for reposongs_song_sequence
Hibernate: select next value for reposongs_song_sequence
```

The post-flush actions insert the rows into the database.

Database Calls Post-Flush

```
Hibernate: insert into reposongs_song (artist, released, title, id) values (?,?,?,?)
Hibernate: insert into reposongs_song (artist, released, title, id) values (?,?,?,?)
Hibernate: insert into reposongs_song (artist, released, title, id) values (?,?,?,?)
Hibernate: insert into reposongs_song (artist, released, title, id) values (?,?,?,?)
Hibernate: insert into reposongs_song (artist, released, title, id) values (?,?,?,?)
```

Call flush() Before Issuing Native SQL Queries

You do not need to call `flush()` in order to eventually have changes written to the

database. However, you must call `flush()` within a transaction to assure that all changes are available to native SQL queries issued against the database. JPA-QL queries will automatically call flush() before executing.

12.3. JpaRepository deleteInBatch

The standard `deleteAll(collection)` will issue deletes one SQL statement at a time as shown in the comments of the following snippet.

```
songsRepository.deleteAll(savedSongs);
//delete from reposongs_song where id=? ①
//delete from reposongs_song where id=?
//delete from reposongs_song where id=?
```

① SQL `DELETE` commands are issues one at a time for each ID

The `JpaRepository.deleteInBatch(collection)` will issue a single DELETE SQL statement with all IDs expressed in the where clause.

```
songsRepository.deleteInBatch(savedSongs);
//delete from reposongs_song where id=? or id=? or id=? ①
```

① one SQL `DELETE` command is issued for all IDs

12.4. JPA References

JPA has the notion of references that represent a promise to an `@Entity` in the database. This is normally done to make loading targeted objects from the database faster and leaving related objects to be accessed only on-demand.

In the following examples, the code is demonstrating how it can form a reference to a persisted object in the database — without going through the overhead of realizing that object.

12.4.1. Reference Exists

In this first example, the referenced object exists and the transaction stays open from the time the reference is created — until the reference was resolved.

Able to Obtain Object through Reference within Active Transaction

```
@Test
@Transactional
void ref_session() {
...
    //when - obtaining a reference with a session
    Song dbSongRef = songsRepository.getReferenceById(song.getId()); ①
    //then
    then(dbSongRef).isNotNull();
```

```
    then(dbSongRef.getId()).isEqualTo(song.getId()); ②
    then(dbSongRef.getTitle()).isEqualTo(song.getTitle()); ③
}
```

① returns only a reference to the `@Entity` — without loading from the database

② still only dealing with the unresolved reference up and to this point

③ actual object resolved from the database at this point

12.4.2. Reference Session Inactive

The following example shows that a reference can only be resolved during its initial transaction. We are able to perform some light commands that can be answered directly from the reference, but as soon as we attempt to access data that would require querying the database — it fails.

Unable to Obtain Object through Reference Outside of Transaction

```
import org.hibernate.LazyInitializationException;
...
@Test
void ref_no_session() {
...
    //when - obtaining a reference without a session
    Song dbSongRef = songsRepository.getReferenceById(song.getId()); ①
    //then - get a reference with basics
    then(dbSongRef).isNotNull();
    then(dbSongRef.getId()).isEqualTo(song.getId()); ②
    assertThatThrownBy(
            () -> dbSongRef.getTitle()) ③
            .isInstanceOf(LazyInitializationException.class);
}
```

① returns only a reference to the `@Entity` from original transaction

② still only dealing with the unresolved reference up and to this point

③ actual object resolution attempted at this point — fails

12.4.3. Bogus Reference

The following example shows that the reference is never attempted to be resolved until something is necessary from the object it represents — beyond its primary key.

Reference Never Resolved until Demand

```
import jakarta.persistence.EntityNotFoundException;
...
@Test
@Transactional
void ref_not_exist() {
    //given
```

```
    int doesNotExist=1234;
    //when
    Song dbSongRef = songsRepository.getReferenceById(doesNotExist); ①
    //then - get a reference with basics
    then(dbSongRef).isNotNull();
    then(dbSongRef.getId()).isEqualTo(doesNotExist); ②
    assertThatThrownBy(
            () -> dbSongRef.getTitle()) ③
            .isInstanceOf(EntityNotFoundException.class);
}
```

① returns only a reference to the `@Entity` with an ID not in database

② still only dealing with the unresolved reference up and to this point

③ actual object resolution attempted at this point — fails

Chapter 13. Custom Queries

Sooner or later, a repository action requires some complexity beyond the ability to leverage a single query-by-example, derived query, or even JPA-QL. We may need to implement some custom logic or may want to encapsulate multiple calls within a single method.

13.1. Custom Query Interface

The following example shows how we can extend the repository interface to implement custom calls using the JPA `EntityManager` and the other repository methods. Our custom implementation will return a random `Song` from the database.

Interface for Public Custom Query Methods

```
public interface SongsRepositoryCustom {
    Optional<Song> random();
}
```

13.2. Repository Extends Custom Query Interface

We then declare the repository to extend the additional custom query interface — making the new method(s) available to callers of the repository.

Repository Implements Custom Query Interface

```
public interface SongsRepository extends JpaRepository<Song, Integer>,
SongsRepositoryCustom { ①
    ...
```

① added additional `SongRepositoryCustom` interface for `SongRepository` to extend

13.3. Custom Query Method Implementation

Of course, the new interface will need an implementation. This will require at least two lower-level database calls

1. determine how many objects there are in the database
2. return a random instance for one of those values

The following snippet shows a portion of the custom method implementation. Note that two additional helper methods are required. We will address them in a moment. By default, this class must have the same name as the interface, followed by "Impl".

Custom Query Method Implementation

```
public class SongsRepositoryCustomImpl implements SongsRepositoryCustom {
    private final SecureRandom random = new SecureRandom();
```

```
...
    @Override
    public Optional<Song> random() {
        Optional randomSong = Optional.empty();
        int count = (int) songsRepository.count(); ①

        if (count!=0) {
            int offset = random.nextInt(count);
            List<Song> songs = songs(offset, 1); ②
            randomSong = songs.isEmpty() ? Optional.empty():Optional.of(songs.get(0));
        }
        return randomSong;
    }
}
```

① leverages `CrudRepository.count()` helper method

② leverages a local, private helper method to access specific `Song`

13.4. Repository Implementation Postfix

If you have an alternate suffix pattern other than "Impl" in your application, you can set that value in an attribute of the `@EnableJpaRepositories` annotation.

The following shows a declaration that sets the suffix to its normal default value (i.e., we did not have to do this). If we changed this postfix value from "Impl" to "Xxx", then we would need to change `SongsRepositoryCustomImpl` to `SongsRepositoryCustomXxx`.

Optional Custom Query Method Implementation Suffix

```
@EnableJpaRepositories(repositoryImplementationPostfix="Impl") ①
```

① `Impl` is the default value. Configure this attribute to use non-Impl postfix

13.5. Helper Methods

The custom `random()` method makes use of two helper methods. One is in the `CrudRepository` interface and the other directly uses the `EntityManager` to issue a query.

CrudRepository.count() Used as Helper Method

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {
    long count();
```

EntityManager NamedQuery used as Helper Method

```
protected List<Song> songs(int offset, int limit) {
    return em.createNamedQuery("Song.songs")
            .setFirstResult(offset)
```

```
            .setMaxResults(limit)
            .getResultList();
}
```

We will need to inject some additional resources to make these calls:

- `SongsRepository`
- `EntityManager`

13.6. Naive Injections

We could have attempted to inject a `SongsRepository` and `EntityManager` straight into the `Impl` class.

Possible Injection Option

```
@RequiredArgsConstructor
public class SongsRepositoryCustomImpl implements SongsRepositoryCustom {
    private final EntityManager em;
    private final SongsRepository songsRepository;
```

However,

- injecting the `EntityManager` would functionally work, but would not necessarily be part of the same Persistence Context and transaction as the rest of the repository
- eagerly injecting the `SongsRepository` in the `Impl` class will not work because the `Impl` class is now part of the `SongsRepository` implementation. We have a recursion problem to resolve there.

13.7. Required Injections

We need to instead

- inject a `JpaContext` and obtain the `EntityManager` from that context
- use `@Autowired @Lazy` and a non-final attribute for the `SongsRepository` injection to indicate that this instance can be initialized without access to the injected bean

Required Injections

```
import org.springframework.data.jpa.repository.JpaContext;
...
public class SongsRepositoryCustomImpl implements SongsRepositoryCustom {
    private final EntityManager em; ①
    @Autowired @Lazy ②
    private SongsRepository songsRepository;

    public SongsRepositoryCustomImpl(JpaContext jpaContext) { ①
        em=jpaContext.getEntityManagerByManagedType(Song.class);
    }
```

① `EntityManager` obtained from injected `JpaContext`

② `SongsRepository` lazily injected to mitigate the recursive dependency between the `Impl` class and the full repository instance

13.8. Calling Custom Query

With all that in place, we can then call our custom `random()` method and obtain a sample `Song` to work with from the database.

Example Custom Query Client Call

```
//when
Optional<Song> randomSong = songsRepository.random();
//then
then(randomSong.isPresent()).isTrue();
```

The following shows the resulting SQL

Custom Random Query Resulting SQL

```
select count(*) from reposongs_song s1_0
select ...
  from reposongs_song s1_0
  offset ? rows fetch first ? rows only
```

Chapter 14. Summary

In this module, we learned:

- that Spring Data JPA eliminates the need to write boilerplate JPA code
- to perform basic CRUD management for `@Entity` classes using a repository
- to implement query-by-example
- unbounded collections can grow over time and cause our applications to eventually fail
  - that paging and sorting can easily be used with repositories
- to implement query methods derived from a query DSL
- to implement custom repository extensions

14.1. Comparing Query Types

Of the query types,

- derived queries and query-by-example are simpler but have their limits
  - derived queries are more expressive
  - query-by-example can be built flexibly at runtime
  - nothing is free — so anything that requires translation between source and JPA form may incur extra initialization and/or processing time
- JPA-QL and native SQL
  - have virtually no limit to what they can express
  - cannot be dynamically defined for a repository like query-by-example. You would need to use the `EntityManager` directly to do that.
  - have loose coupling between the repository method name and the actual function of the executed query
  - can be resolved in an external source file that would allow for query changes without recompiling