

Spring Data JPA - Reference Documentation

Oliver Gierke · Thomas Darimont · Christoph Strobl · Mark Paluch · Jay Bryant – Version 2.1.0.RELEASE,
2018-09-21

© 2008-2018 The original authors.



Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

Table of Contents

Preface

- 1. Project Metadata
- 2. New & Noteworthy
 - 2.1. What's New in Spring Data JPA 1.11
 - 2.2. What's New in Spring Data JPA 1.10
- 3. Dependencies
 - 3.1. Dependency Management with Spring Boot
 - 3.2. Spring Framework
- 4. Working with Spring Data Repositories
 - 4.1. Core concepts
 - 4.2. Query methods
 - 4.3. Defining Repository Interfaces
 - 4.3.1. Fine-tuning Repository Definition
 - 4.3.2. Null Handling of Repository Methods
 - 4.3.3. Using Repositories with Multiple Spring Data Modules
 - 4.4. Defining Query Methods
 - 4.4.1. Query Lookup Strategies
 - 4.4.2. Query Creation
 - 4.4.3. Property Expressions
 - 4.4.4. Special parameter handling
 - 4.4.5. Limiting Query Results
 - 4.4.6. Streaming query results
 - 4.4.7. Async query results
 - 4.5. Creating Repository Instances
 - 4.5.1. XML configuration
 - 4.5.2. JavaConfig
 - 4.5.3. Standalone usage
 - 4.6. Custom Implementations for Spring Data Repositories
 - 4.6.1. Customizing Individual Repositories

4.6.2. Customize the Base Repository

4.7. Publishing Events from Aggregate Roots

4.8. Spring Data Extensions

4.8.1. Querydsl Extension

4.8.2. Web support

4.8.3. Repository Populators

Reference Documentation

5. JPA Repositories

5.1. Introduction

5.1.1. Spring Namespace

5.1.2. Annotation-based Configuration

5.1.3. Bootstrap Mode

5.2. Persisting Entities

5.2.1. Saving Entities

5.3. Query Methods

5.3.1. Query Lookup Strategies

5.3.2. Query Creation

5.3.3. Using JPA Named Queries

5.3.4. Using `@Query`

5.3.5. Using Sort

5.3.6. Using Named Parameters

5.3.7. Using SpEL Expressions

5.3.8. Modifying Queries

5.3.9. Applying Query Hints

5.3.10. Configuring Fetch- and LoadGraphs

5.3.11. Projections

5.4. Stored Procedures

5.5. Specifications

5.6. Query by Example

5.6.1. Introduction

5.6.2. Usage

5.6.3. Example Matchers

5.6.4. Executing an example

5.7. Transactionality

5.7.1. Transactional query methods

5.8. Locking

5.9. Auditing

5.9.1. Basics

5.9.2. JPA Auditing

5.10. Miscellaneous Considerations

5.10.1. Using `JpaContext` in Custom Implementations

5.10.2. Merging persistence units

5.10.3. CDI Integration

Appendix

Appendix A: Namespace reference

The `<repositories />` Element

Appendix B: Populators namespace reference

The `<populator />` element

Appendix C: Repository query keywords

Supported query keywords

Appendix D: Repository query return types

Supported Query Return Types

Appendix E: Frequently Asked Questions

Common

Infrastructure

Auditing

Appendix F: Glossary

Preface

Spring Data JPA provides repository support for the Java Persistence API (JPA). It eases development of applications that need to access JPA data sources.

1. Project Metadata

- Version control - <http://github.com/spring-projects/spring-data-jpa>
- Bugtracker - <https://jira.spring.io/browse/DATAJPA>
- Release repository - <https://repo.spring.io/libs-release>
- Milestone repository - <https://repo.spring.io/libs-milestone>
- Snapshot repository - <https://repo.spring.io/libs-snapshot>

2. New & Noteworthy

2.1. What's New in Spring Data JPA 1.11

Spring Data JPA 1.11 added the following features:

- Improved compatibility with Hibernate 5.2.
- Support any-match mode for [Query by Example](#).
- Paged query execution optimizations.
- Support for the `exists` projection in repository query derivation.

2.2. What's New in Spring Data JPA 1.10

Spring Data JPA 1.10 added the following features:

- Support for [Projections](#) in repository query methods.
- Support for [Query by Example](#).

- The following annotations have been enabled to build on composed annotations: `@EntityGraph`, `@Lock`, `@Modifying`, `@Query`, `@QueryHints`, and `@Procedure`.
- Support for the `Contains` keyword on collection expressions.
- `AttributeConverter` implementations for `ZoneId` of JSR-310 and `ThreeTenBP`.
- Upgrade to Querydsl 4, Hibernate 5, OpenJPA 2.4, and EclipseLink 2.6.1.

3. Dependencies

Due to the different inception dates of individual Spring Data modules, most of them carry different major and minor version numbers. The easiest way to find compatible ones is to rely on the Spring Data Release Train BOM that we ship with the compatible versions defined. In a Maven project, you would declare this dependency in the `<dependencyManagement />` section of your POM, as follows:

Example 1. Using the Spring Data release train BOM

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.data</groupId>
      <artifactId>spring-data-releasetrain</artifactId>
      <version>Lovelace-RELEASE</version>
      <scope>import</scope>
      <type>pom</type>
    </dependency>
  </dependencies>
</dependencyManagement>
```

The current release train version is `Lovelace-RELEASE`. The train names ascend alphabetically and the currently available trains are listed [here](#). The version name follows the following pattern: `${name}-${release}`, where release can be one of the following:

- `BUILD-SNAPSHOT`: Current snapshots
- `M1`, `M2`, and so on: Milestones
- `RC1`, `RC2`, and so on: Release candidates
- `RELEASE`: GA release
- `SR1`, `SR2`, and so on: Service releases

A working example of using the BOMs can be found in our [Spring Data examples repository](#). With that in place, you can declare the Spring Data modules you would like to use without a version in the `<dependencies />` block, as follows:

Example 2. Declaring a dependency to a Spring Data module

```
<dependencies>
  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
```

```
</dependency>
<dependencies>
```

3.1. Dependency Management with Spring Boot

Spring Boot selects a recent version of Spring Data modules for you. If you still want to upgrade to a newer version, configure the property `spring-data-releasetrain.version` to the [train name and iteration](#) you would like to use.

3.2. Spring Framework

The current version of Spring Data modules require Spring Framework in version 5.1.0.RELEASE or better. The modules might also work with an older bugfix version of that minor version. However, using the most recent version within that generation is highly recommended.

§ 4. Working with Spring Data Repositories

The goal of the Spring Data repository abstraction is to significantly reduce the amount of boilerplate code required to implement data access layers for various persistence stores.

Spring Data repository documentation and your module



This chapter explains the core concepts and interfaces of Spring Data repositories. The information in this chapter is pulled from the Spring Data Commons module. It uses the configuration and code samples for the Java Persistence API (JPA) module. You should adapt the XML namespace declaration and the types to be extended to the equivalents of the particular module that you use. “[Namespace reference](#)” covers XML configuration, which is supported across all Spring Data modules supporting the repository API. “[Repository query keywords](#)” covers the query method keywords supported by the repository abstraction in general. For detailed information on the specific features of your module, see the chapter on that module of this document.

4.1. Core concepts

The central interface in the Spring Data repository abstraction is `Repository`. It takes the domain class to manage as well as the ID type of the domain class as type arguments. This interface acts primarily as a marker interface to capture the types to work with and to help you to discover interfaces that extend this one. The `CrudRepository` provides sophisticated CRUD functionality for the entity class that is being managed.

Example 3. CrudRepository interface

```
public interface CrudRepository<T, ID extends Serializable>
    extends Repository<T, ID> {

    <S extends T> S save(S entity);    1
```

```

Optional<T> findById(ID primaryKey); ❷

Iterable<T> findAll();                ❸

long count();                        ❹

void delete(T entity);               ❺

boolean existsById(ID primaryKey);   ❻

// ... more functionality omitted.
}

```

- ❶ Saves the given entity.
- ❷ Returns the entity identified by the given ID.
- ❸ Returns all entities.
- ❹ Returns the number of entities.
- ❺ Deletes the given entity.
- ❻ Indicates whether an entity with the given ID exists.



We also provide persistence technology-specific abstractions, such as `JpaRepository` or `MongoRepository`. Those interfaces extend `CrudRepository` and expose the capabilities of the underlying persistence technology in addition to the rather generic persistence technology-agnostic interfaces such as `CrudRepository`.

On top of the `CrudRepository`, there is a `PagingAndSortingRepository` abstraction that adds additional methods to ease paginated access to entities:

Example 4. `PagingAndSortingRepository` interface

```

public interface PagingAndSortingRepository<T, ID extends Serializable>
    extends CrudRepository<T, ID> {

    Iterable<T> findAll(Sort sort);

    Page<T> findAll(Pageable pageable);

}

```

To access the second page of `User` by a page size of 20, you could do something like the following:

```

PagingAndSortingRepository<User, Long> repository = // ... get access to a bean
Page<User> users = repository.findAll(new PageRequest(1, 20));

```

In addition to query methods, query derivation for both count and delete queries is available. The following list shows the interface definition for a derived count query:

Example 5. Derived Count Query

```
interface UserRepository extends CrudRepository<User, Long> {

    long countByLastname(String lastname);

}
```

The following list shows the interface definition for a derived delete query:

Example 6. Derived Delete Query

```
interface UserRepository extends CrudRepository<User, Long> {

    long deleteByLastname(String lastname);

    List<User> removeByLastname(String lastname);

}
```

4.2. Query methods

Standard CRUD functionality repositories usually have queries on the underlying datastore. With Spring Data, declaring those queries becomes a four-step process:

1. Declare an interface extending Repository or one of its subinterfaces and type it to the domain class and ID type that it should handle, as shown in the following example:

```
interface PersonRepository extends Repository<Person, Long> { ... }
```

2. Declare query methods on the interface.

```
interface PersonRepository extends Repository<Person, Long> {

    List<Person> findByLastname(String lastname);

}
```

3. Set up Spring to create proxy instances for those interfaces, either with [JavaConfig](#) or with [XML configuration](#).

- a. To use Java configuration, create a class similar to the following:

```
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@EnableJpaRepositories
class Config {}
```

- b. To use XML configuration, define a bean similar to the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jpa="http://www.springframework.org/schema/data/jpa"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
```

```
http://www.springframework.org/schema/data/jpa
http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

<jpa:repositories base-package="com.acme.repositories"/>

</beans>
```

The JPA namespace is used in this example. If you use the repository abstraction for any other store, you need to change this to the appropriate namespace declaration of your store module. In other words, you should exchange `jpa` in favor of, for example, `mongodb`.

+ Also, note that the `JavaConfig` variant does not configure a package explicitly, because the package of the annotated class is used by default. To customize the package to scan, use one of the `basePackage...` attributes of the data-store-specific repository's `@Enable${store}Repositories` - annotation.

4. Inject the repository instance and use it, as shown in the following example:

```
class SomeClient {

    private final PersonRepository repository;

    SomeClient(PersonRepository repository) {
        this.repository = repository;
    }

    void doSomething() {
        List<Person> persons = repository.findByLastname("Matthews");
    }
}
```

The sections that follow explain each step in detail:

- [Defining Repository Interfaces](#)
- [Defining Query Methods](#)
- [Creating Repository Instances](#)
- [Custom Implementations for Spring Data Repositories](#)

4.3. Defining Repository Interfaces

First, define a domain class-specific repository interface. The interface must extend `Repository` and be typed to the domain class and an ID type. If you want to expose CRUD methods for that domain type, extend `CrudRepository` instead of `Repository`.

4.3.1. Fine-tuning Repository Definition

Typically, your repository interface extends `Repository`, `CrudRepository`, or `PagingAndSortingRepository`. Alternatively, if you do not want to extend Spring Data interfaces, you can also annotate your repository interface with `@RepositoryDefinition`. Extending `CrudRepository` exposes a complete set of methods to manipulate your entities. If you prefer to be selective about the methods being exposed, copy the methods you want to expose from `CrudRepository` into your domain repository.

Doing so lets you define your own abstractions on top of the provided Spring Data



Repositories functionality.

The following example shows how to selectively expose CRUD methods (`findById` and `save`, in this case):

Example 7. Selectively exposing CRUD methods

```
@NoRepositoryBean
interface MyBaseRepository<T, ID extends Serializable> extends Repository<T, ID> {

    Optional<T> findById(ID id);

    <S extends T> S save(S entity);
}

interface UserRepository extends MyBaseRepository<User, Long> {
    User findByEmailAddress(EmailAddress emailAddress);
}
```

In the prior example, you defined a common base interface for all your domain repositories and exposed `findById(...)` as well as `save(...)`. These methods are routed into the base repository implementation of the store of your choice provided by Spring Data (for example, if you use JPA, the implementation is `SimpleJpaRepository`), because they match the method signatures in `CrudRepository`. So the `UserRepository` can now save users, find individual users by ID, and trigger a query to find `Users` by email address.



The intermediate repository interface is annotated with `@NoRepositoryBean`. Make sure you add that annotation to all repository interfaces for which Spring Data should not create instances at runtime.

4.3.2. Null Handling of Repository Methods

As of Spring Data 2.0, repository CRUD methods that return an individual aggregate instance use Java 8's `Optional` to indicate the potential absence of a value. Besides that, Spring Data supports returning the following wrapper types on query methods:

- `com.google.common.base.Optional`
- `scala.Option`
- `io.vavr.control.Option`
- `javaslang.control.Option` (deprecated as `Javaslang` is deprecated)

Alternatively, query methods can choose not to use a wrapper type at all. The absence of a query result is then indicated by returning `null`. Repository methods returning collections, collection alternatives, wrappers, and streams are guaranteed never to return `null` but rather the corresponding empty representation. See "[Repository query return types](#)" for details.

Nullability Annotations

You can express nullability constraints for repository methods by using [Spring Framework's nullability annotations](#). They provide a tooling-friendly approach and opt-in `null` checks during runtime, as follows:

- `@NonNullApi`: Used on the package level to declare that the default behavior for parameters and return values is to not accept or produce `null` values.
- `@NonNull`: Used on a parameter or return value that must not be `null` (not needed on a parameter and return value where `@NonNullApi` applies).
- `@Nullable`: Used on a parameter or return value that can be `null`.

Spring annotations are meta-annotated with [JSR 305](#) annotations (a dormant but widely spread JSR). JSR 305 meta-annotations let tooling vendors such as [IDEA](#), [Eclipse](#), and [Kotlin](#) provide null-safety support in a generic way, without having to hard-code support for Spring annotations. To enable runtime checking of nullability constraints for query methods, you need to activate non-nullability on the package level by using Spring's `@NonNullApi` in `package-info.java`, as shown in the following example:

Example 8. Declaring Non-nullability in `package-info.java`

```
@org.springframework.lang.NonNullApi
package com.acme;
```

Once non-null defaulting is in place, repository query method invocations get validated at runtime for nullability constraints. If a query execution result violates the defined constraint, an exception is thrown. This happens when the method would return `null` but is declared as non-nullable (the default with the annotation defined on the package the repository resides in). If you want to opt-in to nullable results again, selectively use `@Nullable` on individual methods. Using the result wrapper types mentioned at the start of this section continues to work as expected: An empty result is translated into the value that represents absence.

The following example shows a number of the techniques just described:

Example 9. Using different nullability constraints

```
package com.acme;                                ❶

import org.springframework.lang.Nullable;

interface UserRepository extends Repository<User, Long> {

    User getByEmailAddress(EmailAddress emailAddress);    ❷

    @Nullable
    User findByEmailAddress(@Nullable EmailAddress emailAddress);    ❸

    Optional<User> findOptionalByEmailAddress(EmailAddress emailAddress);    ❹
}
```

❶ The repository resides in a package (or sub-package) for which we have defined non-null behavior.

❷

Throws an `EmptyResultDataAccessException` when the query executed does not produce a result. Throws an `IllegalArgumentException` when the `emailAddress` handed to the method is `null`.

- 3 Returns `null` when the query executed does not produce a result. Also accepts `null` as the value for `emailAddress`.
- 4 Returns `Optional.empty()` when the query executed does not produce a result. Throws an `IllegalArgumentException` when the `emailAddress` handed to the method is `null`.

Nullability in Kotlin-based Repositories

Kotlin has the definition of [nullability constraints](#) baked into the language. Kotlin code compiles to bytecode, which does not express nullability constraints through method signatures but rather through compiled-in metadata. Make sure to include the `kotlin-reflect` JAR in your project to enable introspection of Kotlin's nullability constraints. Spring Data repositories use the language mechanism to define those constraints to apply the same runtime checks, as follows:

Example 10. Using nullability constraints on Kotlin repositories

```
interface UserRepository : Repository<User, String> {
    fun findByUsername(username: String): User      ❶
    fun findByFirstname(firstname: String?): User?  ❷
}
```

The method defines both the parameter and the result as non-nullable (the Kotlin default).

- ❶ The Kotlin compiler rejects method invocations that pass `null` to the method. If the query execution yields an empty result, an `EmptyResultDataAccessException` is thrown.
- ❷ This method accepts `null` for the `firstname` parameter and returns `null` if the query execution does not produce a result.

4.3.3. Using Repositories with Multiple Spring Data Modules

Using a unique Spring Data module in your application makes things simple, because all repository interfaces in the defined scope are bound to the Spring Data module. Sometimes, applications require using more than one Spring Data module. In such cases, a repository definition must distinguish between persistence technologies. When it detects multiple repository factories on the class path, Spring Data enters strict repository configuration mode. Strict configuration uses details on the repository or the domain class to decide about Spring Data module binding for a repository definition:

1. If the repository definition [extends the module-specific repository](#), then it is a valid candidate for the particular Spring Data module.
2. If the domain class is [annotated with the module-specific type annotation](#), then it is a valid candidate for the particular Spring Data module. Spring Data modules accept either third-party annotations (such as JPA's `@Entity`) or provide their own annotations (such as `@Document` for Spring Data MongoDB and Spring Data Elasticsearch).

The following example shows a repository that uses module-specific interfaces (JPA in this case):

Example 11. Repository definitions using module-specific interfaces

```

interface MyRepository extends JpaRepository<User, Long> { }

@NoRepositoryBean
interface MyBaseRepository<T, ID extends Serializable> extends JpaRepository<T, ID> {
    ...
}

interface UserRepository extends MyBaseRepository<User, Long> {
    ...
}

```

`MyRepository` and `UserRepository` extend `JpaRepository` in their type hierarchy. They are valid candidates for the Spring Data JPA module.

The following example shows a repository that uses generic interfaces:

Example 12. Repository definitions using generic interfaces

```

interface AmbiguousRepository extends Repository<User, Long> {
    ...
}

@NoRepositoryBean
interface MyBaseRepository<T, ID extends Serializable> extends CrudRepository<T, ID> {
    ...
}

interface AmbiguousUserRepository extends MyBaseRepository<User, Long> {
    ...
}

```

`AmbiguousRepository` and `AmbiguousUserRepository` extend only `Repository` and `CrudRepository` in their type hierarchy. While this is perfectly fine when using a unique Spring Data module, multiple modules cannot distinguish to which particular Spring Data these repositories should be bound.

The following example shows a repository that uses domain classes with annotations:

Example 13. Repository definitions using domain classes with annotations

```

interface PersonRepository extends Repository<Person, Long> {
    ...
}

@Entity
class Person {
    ...
}

interface UserRepository extends Repository<User, Long> {
    ...
}

@Document

```

```
class User {
    ...
}
```

`PersonRepository` references `Person`, which is annotated with the JPA `@Entity` annotation, so this repository clearly belongs to Spring Data JPA. `UserRepository` references `User`, which is annotated with Spring Data MongoDB's `@Document` annotation.

The following bad example shows a repository that uses domain classes with mixed annotations:

Example 14. Repository definitions using domain classes with mixed annotations

```
interface JpaPersonRepository extends Repository<Person, Long> {
    ...
}

interface MongoDBPersonRepository extends Repository<Person, Long> {
    ...
}

@Entity
@Document
class Person {
    ...
}
```

This example shows a domain class using both JPA and Spring Data MongoDB annotations. It defines two repositories, `JpaPersonRepository` and `MongoDBPersonRepository`. One is intended for JPA and the other for MongoDB usage. Spring Data is no longer able to tell the repositories apart, which leads to undefined behavior.

[Repository type details](#) and [distinguishing domain class annotations](#) are used for strict repository configuration to identify repository candidates for a particular Spring Data module. Using multiple persistence technology-specific annotations on the same domain type is possible and enables reuse of domain types across multiple persistence technologies. However, Spring Data can then no longer determine a unique module with which to bind the repository.

The last way to distinguish repositories is by scoping repository base packages. Base packages define the starting points for scanning for repository interface definitions, which implies having repository definitions located in the appropriate packages. By default, annotation-driven configuration uses the package of the configuration class. The [base package in XML-based configuration](#) is mandatory.

The following example shows annotation-driven configuration of base packages:

Example 15. Annotation-driven configuration of base packages

```
@EnableJpaRepositories(basePackages = "com.acme.repositories.jpa")
@EnableMongoRepositories(basePackages = "com.acme.repositories.mongo")
interface Configuration { }
```

4.4. Defining Query Methods

The repository proxy has two ways to derive a store-specific query from the method name:

- By deriving the query from the method name directly.
- By using a manually defined query.

Available options depend on the actual store. However, there must be a strategy that decides what actual query is created. The next section describes the available options.

4.4.1. Query Lookup Strategies

The following strategies are available for the repository infrastructure to resolve the query. With XML configuration, you can configure the strategy at the namespace through the `query-lookup-strategy` attribute. For Java configuration, you can use the `queryLookupStrategy` attribute of the `Enable${store}Repositories` annotation. Some strategies may not be supported for particular datastores.

- `CREATE` attempts to construct a store-specific query from the query method name. The general approach is to remove a given set of well known prefixes from the method name and parse the rest of the method. You can read more about query construction in [“Query Creation”](#).
- `USE_DECLARED_QUERY` tries to find a declared query and throws an exception if cannot find one. The query can be defined by an annotation somewhere or declared by other means. Consult the documentation of the specific store to find available options for that store. If the repository infrastructure does not find a declared query for the method at bootstrap time, it fails.
- `CREATE_IF_NOT_FOUND` (default) combines `CREATE` and `USE_DECLARED_QUERY`. It looks up a declared query first, and, if no declared query is found, it creates a custom method name-based query. This is the default lookup strategy and, thus, is used if you do not configure anything explicitly. It allows quick query definition by method names but also custom-tuning of these queries by introducing declared queries as needed.

4.4.2. Query Creation

The query builder mechanism built into Spring Data repository infrastructure is useful for building constraining queries over entities of the repository. The mechanism strips the prefixes `find...By`, `read...By`, `query...By`, `count...By`, and `get...By` from the method and starts parsing the rest of it. The introducing clause can contain further expressions, such as a `Distinct` to set a distinct flag on the query to be created. However, the first `By` acts as delimiter to indicate the start of the actual criteria. At a very basic level, you can define conditions on entity properties and concatenate them with `And` and `Or`. The following example shows how to create a number of queries:

Example 16. Query creation from method names

```
interface PersonRepository extends Repository<User, Long> {  
  
    List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);  
  
    // Enables the distinct flag for the query  
    List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);  
    List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);  
  
    // Enabling ignoring case for an individual property
```

```

List<Person> findByLastnameIgnoreCase(String lastname);
// Enabling ignoring case for all suitable properties
List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

// Enabling static ORDER BY for a query
List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
}

```

The actual result of parsing the method depends on the persistence store for which you create the query. However, there are some general things to notice:

- The expressions are usually property traversals combined with operators that can be concatenated. You can combine property expressions with `AND` and `OR`. You also get support for operators such as `Between`, `LessThan`, `GreaterThan`, and `Like` for the property expressions. The supported operators can vary by datastore, so consult the appropriate part of your reference documentation.
- The method parser supports setting an `IgnoreCase` flag for individual properties (for example, `findByLastnameIgnoreCase(...)`) or for all properties of a type that supports ignoring case (usually `String` instances — for example, `findByLastnameAndFirstnameAllIgnoreCase(...)`). Whether ignoring cases is supported may vary by store, so consult the relevant sections in the reference documentation for the store-specific query method.
- You can apply static ordering by appending an `OrderBy` clause to the query method that references a property and by providing a sorting direction (`Asc` or `Desc`). To create a query method that supports dynamic sorting, see [“Special parameter handling”](#).

4.4.3. Property Expressions

Property expressions can refer only to a direct property of the managed entity, as shown in the preceding example. At query creation time, you already make sure that the parsed property is a property of the managed domain class. However, you can also define constraints by traversing nested properties. Consider the following method signature:

```
List<Person> findByAddressZipCode(ZipCode zipCode);
```

Assume a `Person` has an `Address` with a `ZipCode`. In that case, the method creates the property traversal `x.address.zipCode`. The resolution algorithm starts by interpreting the entire part (`AddressZipCode`) as the property and checks the domain class for a property with that name (uncapitalized). If the algorithm succeeds, it uses that property. If not, the algorithm splits up the source at the camel case parts from the right side into a head and a tail and tries to find the corresponding property — in our example, `AddressZip` and `Code`. If the algorithm finds a property with that head, it takes the tail and continues building the tree down from there, splitting the tail up in the way just described. If the first split does not match, the algorithm moves the split point to the left (`Address`, `ZipCode`) and continues.

Although this should work for most cases, it is possible for the algorithm to select the wrong property. Suppose the `Person` class has an `addressZip` property as well. The algorithm would match in the first split round already, choose the wrong property, and fail (as the type of `addressZip` probably has no `code` property).

To resolve this ambiguity you can use `_` inside your method name to manually define traversal points. So our method name would be as follows:

```
List<Person> findByAddress_ZipCode(ZipCode zipCode);
```

Because we treat the underscore character as a reserved character, we strongly advise following standard Java naming conventions (that is, not using underscores in property names but using camel case instead).

4.4.4. Special parameter handling

To handle parameters in your query, define method parameters as already seen in the preceding examples. Besides that, the infrastructure recognizes certain specific types like `Pageable` and `Sort`, to apply pagination and sorting to your queries dynamically. The following example demonstrates these features:

Example 17. Using `Pageable`, `Slice`, and `Sort` in query methods

```
Page<User> findByLastname(String lastname, Pageable pageable);

Slice<User> findByLastname(String lastname, Pageable pageable);

List<User> findByLastname(String lastname, Sort sort);

List<User> findByLastname(String lastname, Pageable pageable);
```

The first method lets you pass an `org.springframework.data.domain.Pageable` instance to the query method to dynamically add paging to your statically defined query. A `Page` knows about the total number of elements and pages available. It does so by the infrastructure triggering a count query to calculate the overall number. As this might be expensive (depending on the store used), you can instead return a `Slice`. A `Slice` only knows about whether a next `Slice` is available, which might be sufficient when walking through a larger result set.

Sorting options are handled through the `Pageable` instance, too. If you only need sorting, add an `org.springframework.data.domain.Sort` parameter to your method. As you can see, returning a `List` is also possible. In this case, the additional metadata required to build the actual `Page` instance is not created (which, in turn, means that the additional count query that would have been necessary is not issued). Rather, it restricts the query to look up only the given range of entities.



To find out how many pages you get for an entire query, you have to trigger an additional count query. By default, this query is derived from the query you actually trigger.

4.4.5. Limiting Query Results

The results of query methods can be limited by using the `first` or `top` keywords, which can be used interchangeably. An optional numeric value can be appended to `top` or `first` to specify the maximum result size to be returned. If the number is left out, a result size of 1 is assumed. The following example shows how to limit the query size:

Example 18. Limiting the result size of a query with `Top` and `First`

```
User findFirstByOrderByLastnameAsc();

User findTopByOrderByAgeDesc();

Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);

Slice<User> findTop3ByLastname(String lastname, Pageable pageable);

List<User> findFirst10ByLastname(String lastname, Sort sort);

List<User> findTop10ByLastname(String lastname, Pageable pageable);
```

The limiting expressions also support the `Distinct` keyword. Also, for the queries limiting the result set to one instance, wrapping the result into with the `Optional` keyword is supported.

If pagination or slicing is applied to a limiting query pagination (and the calculation of the number of pages available), it is applied within the limited result.



Limiting the results in combination with dynamic sorting by using a `Sort` parameter lets you express query methods for the 'K' smallest as well as for the 'K' biggest elements.

4.4.6. Streaming query results

The results of query methods can be processed incrementally by using a Java 8 `Stream<T>` as return type. Instead of wrapping the query results in a `Stream` data store-specific methods are used to perform the streaming, as shown in the following example:

Example 19. Stream the result of a query with Java 8 `Stream<T>`

```
@Query("select u from User u")
Stream<User> findAllByCustomQueryAndStream();

Stream<User> readAllByFirstnameNotNull();

@Query("select u from User u")
Stream<User> streamAllPaged(Pageable pageable);
```



A `Stream` potentially wraps underlying data store-specific resources and must, therefore, be closed after usage. You can either manually close the `Stream` by using the `close()` method or by using a Java 7 `try-with-resources` block, as shown in the following example:

Example 20. Working with a `Stream<T>` result in a `try-with-resources` block

```
try (Stream<User> stream = repository.findAllByCustomQueryAndStream()) {  
    stream.forEach(...);  
}
```



Not all Spring Data modules currently support `Stream<T>` as a return type.

4.4.7. Async query results

Repository queries can be run asynchronously by using [Spring's asynchronous method execution capability](#). This means the method returns immediately upon invocation while the actual query execution occurs in a task that has been submitted to a Spring `TaskExecutor`. Asynchronous query execution is different from reactive query execution and should not be mixed. Refer to store-specific documentation for more details on reactive support. The following example shows a number of asynchronous queries:

```
@Async  
Future<User> findByFirstname(String firstname); ❶  
  
@Async  
CompletableFuture<User> findOneByFirstname(String firstname); ❷  
  
@Async  
ListenableFuture<User> findOneByLastname(String lastname); ❸
```

- ❶ Use `java.util.concurrent.Future` as the return type.
- ❷ Use a Java 8 `java.util.concurrent.CompletableFuture` as the return type.
- ❸ Use a `org.springframework.util.concurrent.ListenableFuture` as the return type.

4.5. Creating Repository Instances

In this section, you create instances and bean definitions for the defined repository interfaces. One way to do so is by using the Spring namespace that is shipped with each Spring Data module that supports the repository mechanism, although we generally recommend using Java configuration.

4.5.1. XML configuration

Each Spring Data module includes a `repositories` element that lets you define a base package that Spring scans for you, as shown in the following example:

Example 21. Enabling Spring Data repositories via XML

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xmlns="http://www.springframework.org/schema/data/jpa"  
    xsi:schemaLocation="http://www.springframework.org/schema/beans  
        http://www.springframework.org/schema/beans/spring-beans.xsd  
        http://www.springframework.org/schema/data/jpa
```

```
http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">  
  
    <repositories base-package="com.acme.repositories" />  
  
</beans:beans>
```

In the preceding example, Spring is instructed to scan `com.acme.repositories` and all its sub-packages for interfaces extending `Repository` or one of its sub-interfaces. For each interface found, the infrastructure registers the persistence technology-specific `FactoryBean` to create the appropriate proxies that handle invocations of the query methods. Each bean is registered under a bean name that is derived from the interface name, so an interface of `UserRepository` would be registered under `userRepository`. The `base-package` attribute allows wildcards so that you can define a pattern of scanned packages.

Using filters

By default, the infrastructure picks up every interface extending the persistence technology-specific `Repository` sub-interface located under the configured base package and creates a bean instance for it. However, you might want more fine-grained control over which interfaces have bean instances created for them. To do so, use `<include-filter />` and `<exclude-filter />` elements inside the `<repositories />` element. The semantics are exactly equivalent to the elements in Spring's context namespace. For details, see the [Spring reference documentation](#) for these elements.

For example, to exclude certain interfaces from instantiation as repository beans, you could use the following configuration:

Example 22. Using exclude-filter element

```
<repositories base-package="com.acme.repositories">  
    <context:exclude-filter type="regex" expression=".*SomeRepository" />  
</repositories>
```

The preceding example excludes all interfaces ending in `SomeRepository` from being instantiated.

4.5.2. JavaConfig

The repository infrastructure can also be triggered by using a store-specific `@Enable${store}Repositories` annotation on a JavaConfig class. For an introduction into Java-based configuration of the Spring container, see [JavaConfig in the Spring reference documentation](#).

A sample configuration to enable Spring Data repositories resembles the following:

Example 23. Sample annotation based repository configuration

```
@Configuration  
@EnableJpaRepositories("com.acme.repositories")  
class ApplicationConfiguration {  
  
    @Bean  
    EntityManagerFactory entityManagerFactory() {  
        // ...  
    }  
}
```

```
}  
}
```



The preceding example uses the JPA-specific annotation, which you would change according to the store module you actually use. The same applies to the definition of the `EntityManagerFactory` bean. See the sections covering the store-specific configuration.

4.5.3. Standalone usage

You can also use the repository infrastructure outside of a Spring container — for example, in CDI environments. You still need some Spring libraries in your classpath, but, generally, you can set up repositories programmatically as well. The Spring Data modules that provide repository support ship a persistence technology-specific `RepositoryFactory` that you can use as follows:

Example 24. Standalone usage of repository factory

```
RepositoryFactorySupport factory = ... // Instantiate factory here  
UserRepository repository = factory.getRepository(UserRepository.class);
```

4.6. Custom Implementations for Spring Data Repositories

This section covers repository customization and how fragments form a composite repository.

When a query method requires a different behavior or cannot be implemented by query derivation, then it is necessary to provide a custom implementation. Spring Data repositories let you provide custom repository code and integrate it with generic CRUD abstraction and query method functionality.

4.6.1. Customizing Individual Repositories

To enrich a repository with custom functionality, you must first define a fragment interface and an implementation for the custom functionality, as shown in the following example:

Example 25. Interface for custom repository functionality

```
interface CustomizedUserRepository {  
    void someCustomMethod(User user);  
}
```

Then you can let your repository interface additionally extend from the fragment interface, as shown in the following example:

Example 26. Implementation of custom repository functionality

```
class CustomizedUserRepositoryImpl implements CustomizedUserRepository {  
    public void someCustomMethod(User user) {
```

```
// Your custom implementation
}
}
```



The most important part of the class name that corresponds to the fragment interface is the `Impl` postfix.

The implementation itself does not depend on Spring Data and can be a regular Spring bean. Consequently, you can use standard dependency injection behavior to inject references to other beans (such as a `JdbcTemplate`), take part in aspects, and so on.

You can let your repository interface extend the fragment interface, as shown in the following example:

Example 27. Changes to your repository interface

```
interface UserRepository extends CrudRepository<User, Long>, CustomizedUserRepository {

    // Declare query methods here
}
```

Extending the fragment interface with your repository interface combines the CRUD and custom functionality and makes it available to clients.

Spring Data repositories are implemented by using fragments that form a repository composition. Fragments are the base repository, functional aspects (such as [QueryDsl](#)), and custom interfaces along with their implementation. Each time you add an interface to your repository interface, you enhance the composition by adding a fragment. The base repository and repository aspect implementations are provided by each Spring Data module.

The following example shows custom interfaces and their implementations:

Example 28. Fragments with their implementations

```
interface HumanRepository {
    void someHumanMethod(User user);
}

class HumanRepositoryImpl implements HumanRepository {

    public void someHumanMethod(User user) {
        // Your custom implementation
    }
}

interface ContactRepository {

    void someContactMethod(User user);

    User anotherContactMethod(User user);
}
```

```

}

class ContactRepositoryImpl implements ContactRepository {

    public void someContactMethod(User user) {
        // Your custom implementation
    }

    public User anotherContactMethod(User user) {
        // Your custom implementation
    }
}

```

The following example shows the interface for a custom repository that extends `CrudRepository`:

Example 29. Changes to your repository interface

```

interface UserRepository extends CrudRepository<User, Long>, HumanRepository, ContactRepository {

    // Declare query methods here
}

```

Repositories may be composed of multiple custom implementations that are imported in the order of their declaration. Custom implementations have a higher priority than the base implementation and repository aspects. This ordering lets you override base repository and aspect methods and resolves ambiguity if two fragments contribute the same method signature. Repository fragments are not limited to use in a single repository interface. Multiple repositories may use a fragment interface, letting you reuse customizations across different repositories.

The following example shows a repository fragment and its implementation:

Example 30. Fragments overriding `save(...)`

```

interface CustomizedSave<T> {
    <S extends T> S save(S entity);
}

class CustomizedSaveImpl<T> implements CustomizedSave<T> {

    public <S extends T> S save(S entity) {
        // Your custom implementation
    }
}

```

The following example shows a repository that uses the preceding repository fragment:

Example 31. Customized repository interfaces

```

interface UserRepository extends CrudRepository<User, Long>, CustomizedSave<User> {
}

```

```
interface PersonRepository extends CrudRepository<Person, Long>, CustomizedSave<Person> {  
}
```

Configuration

If you use namespace configuration, the repository infrastructure tries to autodetect custom implementation fragments by scanning for classes below the package in which it found a repository. These classes need to follow the naming convention of appending the namespace element's `repository-impl-postfix` attribute to the fragment interface name. This postfix defaults to `Impl`. The following example shows a repository that uses the default postfix and a repository that sets a custom value for the postfix:

Example 32. Configuration example

```
<repositories base-package="com.acme.repository" />  
  
<repositories base-package="com.acme.repository" repository-impl-postfix="MyPostfix" />
```

The first configuration in the preceding example tries to look up a class called `com.acme.repository.CustomizedUserRepositoryImpl` to act as a custom repository implementation. The second example tries to lookup `com.acme.repository.CustomizedUserRepositoryMyPostfix`.

Resolution of Ambiguity

If multiple implementations with matching class names are found in different packages, Spring Data uses the bean names to identify which one to use.

Given the following two custom implementations for the `CustomizedUserRepository` shown earlier, the first implementation is used. Its bean name is `customizedUserRepositoryImpl`, which matches that of the fragment interface (`CustomizedUserRepository`) plus the postfix `Impl`.

Example 33. Resolution of ambiguous implementations

```
package com.acme.impl.one;  
  
class CustomizedUserRepositoryImpl implements CustomizedUserRepository {  
  
    // Your custom implementation  
}
```

```
package com.acme.impl.two;  
  
@Component("specialCustomImpl")  
class CustomizedUserRepositoryImpl implements CustomizedUserRepository {  
  
    // Your custom implementation  
}
```

If you annotate the `UserRepository` interface with `@Component("specialCustom")`, the bean name plus `Impl` then matches the one defined for the repository implementation in `com.acme.impl.two`, and it is

used instead of the first one.

Manual Wiring

If your custom implementation uses annotation-based configuration and autowiring only, the preceding approach shown works well, because it is treated as any other Spring bean. If your implementation fragment bean needs special wiring, you can declare the bean and name it according to the conventions described in the [preceding section](#). The infrastructure then refers to the manually defined bean definition by name instead of creating one itself. The following example shows how to manually wire a custom implementation:

Example 34. Manual wiring of custom implementations

```
<repositories base-package="com.acme.repository" />

<beans:bean id="userRepositoryImpl" class="...">
  <!-- further configuration -->
</beans:bean>
```

4.6.2. Customize the Base Repository

The approach described in the [preceding section](#) requires customization of each repository interfaces when you want to customize the base repository behavior so that all repositories are affected. To instead change behavior for all repositories, you can create an implementation that extends the persistence technology-specific repository base class. This class then acts as a custom base class for the repository proxies, as shown in the following example:

Example 35. Custom repository base class

```
class MyRepositoryImpl<T, ID extends Serializable>
  extends SimpleJpaRepository<T, ID> {

  private final EntityManager entityManager;

  MyRepositoryImpl(JpaEntityInformation entityInformation,
                  EntityManager entityManager) {
    super(entityInformation, entityManager);

    // Keep the EntityManager around to used from the newly introduced methods.
    this.entityManager = entityManager;
  }

  @Transactional
  public <S extends T> S save(S entity) {
    // implementation goes here
  }
}
```



The class needs to have a constructor of the super class which the store-specific repository factory implementation uses. If the repository base class has multiple constructors, override the one taking an `EntityInformation` plus a store specific infrastructure object (such as an `EntityManager` or a template class).

The final step is to make the Spring Data infrastructure aware of the customized repository base class. In Java configuration, you can do so by using the `repositoryBaseClass` attribute of the `@Enable${store}Repositories` annotation, as shown in the following example:

Example 36. Configuring a custom repository base class using JavaConfig

```
@Configuration
@EnableJpaRepositories(repositoryBaseClass = MyRepositoryImpl.class)
class ApplicationConfiguration { ... }
```

A corresponding attribute is available in the XML namespace, as shown in the following example:

Example 37. Configuring a custom repository base class using XML

```
<repositories base-package="com.acme.repository"
  base-class="...MyRepositoryImpl" />
```

4.7. Publishing Events from Aggregate Roots

Entities managed by repositories are aggregate roots. In a Domain-Driven Design application, these aggregate roots usually publish domain events. Spring Data provides an annotation called `@DomainEvents` that you can use on a method of your aggregate root to make that publication as easy as possible, as shown in the following example:

Example 38. Exposing domain events from an aggregate root

```
class AnAggregateRoot {

    @DomainEvents ❶
    Collection<Object> domainEvents() {
        // ... return events you want to get published here
    }

    @AfterDomainEventPublication ❷
    void callbackMethod() {
        // ... potentially clean up domain events list
    }

}
```

- ❶ The method using `@DomainEvents` can return either a single event instance or a collection of events. It must not take any arguments.

After all events have been published, we have a method annotated with

- ❷ `@AfterDomainEventPublication`. It can be used to potentially clean the list of events to be published (among other uses).

The methods are called every time one of a Spring Data repository's `save(...)` methods is called.

4.8. Spring Data Extensions

This section documents a set of Spring Data extensions that enable Spring Data usage in a variety of contexts. Currently, most of the integration is targeted towards Spring MVC.

4.8.1. Querydsl Extension

[Querydsl](#) is a framework that enables the construction of statically typed SQL-like queries through its fluent API.

Several Spring Data modules offer integration with Querydsl through `QuerydslPredicateExecutor`, as shown in the following example:

Example 39. QuerydslPredicateExecutor interface

```
public interface QuerydslPredicateExecutor<T> {  
  
    Optional<T> findById(Predicate predicate); ❶  
  
    Iterable<T> findAll(Predicate predicate); ❷  
  
    long count(Predicate predicate);           ❸  
  
    boolean exists(Predicate predicate);       ❹  
  
    // ... more functionality omitted.  
}
```

- ❶ Finds and returns a single entity matching the `Predicate`.
- ❷ Finds and returns all entities matching the `Predicate`.
- ❸ Returns the number of entities matching the `Predicate`.
- ❹ Returns whether an entity that matches the `Predicate` exists.

To make use of Querydsl support, extend `QuerydslPredicateExecutor` on your repository interface, as shown in the following example

Example 40. Querydsl integration on repositories

```
interface UserRepository extends CrudRepository<User, Long>, QuerydslPredicateExecutor<User> {  
}
```

The preceding example lets you write typesafe queries using Querydsl `Predicate` instances, as shown in the following example:

```
Predicate predicate = user.firstname.equalsIgnoreCase("dave")  
    .and(user.lastname.startsWithIgnoreCase("mathews"));  
  
userRepository.findAll(predicate);
```

4.8.2. Web support



This section contains the documentation for the Spring Data web support as it is implemented in the current (and later) versions of Spring Data Commons. As the newly introduced support changes many things, we kept the documentation of the former behavior in [\[web.legacy\]](#).

Spring Data modules that support the repository programming model ship with a variety of web support. The web related components require Spring MVC JARs to be on the classpath. Some of them even provide integration with [Spring HATEOAS](#). In general, the integration support is enabled by using the `@EnableSpringDataWebSupport` annotation in your JavaConfig configuration class, as shown in the following example:

Example 41. Enabling Spring Data web support

```
@Configuration
@EnableWebMvc
@EnableSpringDataWebSupport
class WebConfiguration {}
```

The `@EnableSpringDataWebSupport` annotation registers a few components we will discuss in a bit. It will also detect Spring HATEOAS on the classpath and register integration components for it as well if present.

Alternatively, if you use XML configuration, register either `SpringDataWebConfiguration` or `HateoasAwareSpringDataWebConfiguration` as Spring beans, as shown in the following example (for `SpringDataWebConfiguration`):

Example 42. Enabling Spring Data web support in XML

```
<bean class="org.springframework.data.web.config.SpringDataWebConfiguration" />

<!-- If you use Spring HATEOAS, register this one *instead* of the former -->
<bean class="org.springframework.data.web.config.HateoasAwareSpringDataWebConfiguration" />
```

Basic Web Support

The configuration shown in the [previous section](#) registers a few basic components:

- A `DomainClassConverter` to let Spring MVC resolve instances of repository-managed domain classes from request parameters or path variables.
- `HandlerMethodArgumentResolver` implementations to let Spring MVC resolve `Pageable` and `Sort` instances from request parameters.

DomainClassConverter

The `DomainClassConverter` lets you use domain types in your Spring MVC controller method signatures directly, so that you need not manually lookup the instances through the repository, as shown in the following example:

Example 43. A Spring MVC controller using domain types in method signatures

```

@Controller
@RequestMapping("/users")
class UserController {

    @RequestMapping("/{id}")
    String showUserForm(@PathVariable("id") User user, Model model) {

        model.addAttribute("user", user);
        return "userForm";
    }
}

```

As you can see, the method receives a `User` instance directly, and no further lookup is necessary. The instance can be resolved by letting Spring MVC convert the path variable into the `id` type of the domain class first and eventually access the instance through calling `findById(...)` on the repository instance registered for the domain type.



Currently, the repository has to implement `CrudRepository` to be eligible to be discovered for conversion.

HandlerMethodArgumentResolvers for Pageable and Sort

The configuration snippet shown in the [previous section](#) also registers a

`PageableHandlerMethodArgumentResolver` as well as an instance of `SortHandlerMethodArgumentResolver`. The registration enables `Pageable` and `Sort` as valid controller method arguments, as shown in the following example:

Example 44. Using Pageable as controller method argument

```

@Controller
@RequestMapping("/users")
class UserController {

    private final UserRepository repository;

    UserController(UserRepository repository) {
        this.repository = repository;
    }

    @RequestMapping
    String showUsers(Model model, Pageable pageable) {

        model.addAttribute("users", repository.findAll(pageable));
        return "users";
    }
}

```

The preceding method signature causes Spring MVC try to derive a `Pageable` instance from the request parameters by using the following default configuration:

Table 1. Request parameters evaluated for `Pageable` instances

page	Page you want to retrieve. 0-indexed and defaults to 0.
size	Size of the page you want to retrieve. Defaults to 20.
sort	Properties that should be sorted by in the format <code>property,property(,ASC DESC)</code> . Default sort direction is ascending. Use multiple <code>sort</code> parameters if you want to switch directions — for example, <code>?sort=firstname&sort=lastname,asc</code> .

To customize this behavior, register a bean implementing the `PageableHandlerMethodArgumentResolverCustomizer` interface or the `SortHandlerMethodArgumentResolverCustomizer` interface, respectively. Its `customize()` method gets called, letting you change settings, as shown in the following example:

```
@Bean SortHandlerMethodArgumentResolverCustomizer sortCustomizer() {
    return s -> s.setPropertyDelimiter("<-->");
}
```

If setting the properties of an existing `MethodArgumentResolver` is not sufficient for your purpose, extend either `SpringDataWebConfiguration` or the HATEOAS-enabled equivalent, override the `pageableResolver()` or `sortResolver()` methods, and import your customized configuration file instead of using the `@Enable` annotation.

If you need multiple `Pageable` or `Sort` instances to be resolved from the request (for multiple tables, for example), you can use Spring's `@Qualifier` annotation to distinguish one from another. The request parameters then have to be prefixed with `${qualifier}_`. The following example shows the resulting method signature:

```
String showUsers(Model model,
    @Qualifier("thing1") Pageable first,
    @Qualifier("thing2") Pageable second) { ... }
```

you have to populate `thing1_page` and `thing2_page` and so on.

The default `Pageable` passed into the method is equivalent to a new `PageRequest(0, 20)` but can be customized by using the `@PageableDefault` annotation on the `Pageable` parameter.

Hypermedia Support for Pageables

Spring HATEOAS ships with a representation model class (`PagedResources`) that allows enriching the content of a `Page` instance with the necessary `Page` metadata as well as links to let the clients easily navigate the pages. The conversion of a `Page` to a `PagedResources` is done by an implementation of the Spring HATEOAS `ResourceAssembler` interface, called the `PagedResourcesAssembler`. The following example shows how to use a `PagedResourcesAssembler` as a controller method argument:

Example 45. Using a `PagedResourcesAssembler` as controller method argument

```
@Controller
class PersonController {
```

```

@Autowired PersonRepository repository;

@RequestMapping(value = "/persons", method = RequestMethod.GET)
HttpEntity<PagedResources<Person>> persons(Pageable pageable,
    PagedResourcesAssembler assembler) {

    Page<Person> persons = repository.findAll(pageable);
    return new ResponseEntity<>(assembler.toResources(persons), HttpStatus.OK);
}

```

Enabling the configuration as shown in the preceding example lets the `PagedResourcesAssembler` be used as a controller method argument. Calling `toResources(...)` on it has the following effects:

- The content of the `Page` becomes the content of the `PagedResources` instance.
- The `PagedResources` object gets a `PageMetadata` instance attached, and it is populated with information from the `Page` and the underlying `PageRequest`.
- The `PagedResources` may get `prev` and `next` links attached, depending on the page's state. The links point to the URI to which the method maps. The pagination parameters added to the method match the setup of the `PageableHandlerMethodArgumentResolver` to make sure the links can be resolved later.

Assume we have 30 `Person` instances in the database. You can now trigger a request (GET <http://localhost:8080/persons>) and see output similar to the following:

```

{ "links" : [ { "rel" : "next",
                "href" : "http://localhost:8080/persons?page=1&size=20" }
            ],
  "content" : [
    ... // 20 Person instances rendered here
  ],
  "pageMetadata" : {
    "size" : 20,
    "totalElements" : 30,
    "totalPages" : 2,
    "number" : 0
  }
}

```

You see that the assembler produced the correct URI and also picked up the default configuration to resolve the parameters into a `Pageable` for an upcoming request. This means that, if you change that configuration, the links automatically adhere to the change. By default, the assembler points to the controller method it was invoked in, but that can be customized by handing in a custom `Link` to be used as base to build the pagination links, which overloads the `PagedResourcesAssembler.toResource(...)` method.

Web Databinding Support

Spring Data projections (described in [Projections](#)) can be used to bind incoming request payloads by either using [JSONPath](#) expressions (requires [Jayway JsonPath](#) or [XPath](#) expressions (requires [XmlBeam](#)), as shown in the following example:

Example 46. HTTP payload binding using JSONPath or XPath expressions

```
@ProjectedPayload
public interface UserPayload {

    @XBRead("//firstname")
    @JsonPath("$.firstname")
    String getFirstname();

    @XBRead("/lastname")
    @JsonPath({ "$.lastname", "$.user.lastname" })
    String getLastName();
}
```

The type shown in the preceding example can be used as a Spring MVC handler method argument or by using `ParameterizedTypeReference` on one of `RestTemplate`'s methods. The preceding method declarations would try to find `firstname` anywhere in the given document. The `lastname` XML lookup is performed on the top-level of the incoming document. The JSON variant of that tries a top-level `lastname` first but also tries `lastname` nested in a `user` sub-document if the former does not return a value. That way, changes in the structure of the source document can be mitigated easily without having clients calling the exposed methods (usually a drawback of class-based payload binding).

Nested projections are supported as described in [Projections](#). If the method returns a complex, non-interface type, a Jackson `ObjectMapper` is used to map the final value.

For Spring MVC, the necessary converters are registered automatically as soon as

`@EnableSpringDataWebSupport` is active and the required dependencies are available on the classpath. For usage with `RestTemplate`, register a `ProjectingJackson2HttpMessageConverter` (JSON) or `XmlBeamHttpMessageConverter` manually.

For more information, see the [web projection example](#) in the canonical [Spring Data Examples repository](#).

Querydsl Web Support

For those stores having [QueryDSL](#) integration, it is possible to derive queries from the attributes contained in a `Request` query string.

Consider the following query string:

```
?firstname=Dave&lastname=Matthews
```

Given the `User` object from previous examples, a query string can be resolved to the following value by using the `QuerydslPredicateArgumentResolver`.

```
QUser.user.firstname.eq("Dave").and(QUser.user.lastname.eq("Matthews"))
```



The feature is automatically enabled, along with `@EnableSpringDataWebSupport`, when `Querydsl` is found on the classpath.

Adding a `@QuerydslPredicate` to the method signature provides a ready-to-use `Predicate`, which can be run by using the `QuerydslPredicateExecutor`.



Type information is typically resolved from the method's return type. Since that information does not necessarily match the domain type, it might be a good idea to use the `root` attribute of `QuerydslPredicate`.

The following example shows how to use `@QuerydslPredicate` in a method signature:

```
@Controller
class UserController {

    @Autowired UserRepository repository;

    @RequestMapping(value = "/", method = RequestMethod.GET)
    String index(Model model, @QuerydslPredicate(root = User.class) Predicate predicate, ❶
        Pageable pageable, @RequestParam MultiValueMap<String, String> parameters) {

        model.addAttribute("users", repository.findAll(predicate, pageable));

        return "index";
    }
}
```

❶ Resolve query string arguments to matching `Predicate` for `User`.

The default binding is as follows:

- `Object` on simple properties as `eq`.
- `Object` on collection like properties as `contains`.
- `Collection` on simple properties as `in`.

Those bindings can be customized through the `bindings` attribute of `@QuerydslPredicate` or by making use of Java 8 default methods and adding the `QuerydslBinderCustomizer` method to the repository interface.

```
interface UserRepository extends CrudRepository<User, String>,
    QuerydslPredicateExecutor<User>, ❶
    QuerydslBinderCustomizer<QUser> { ❷

    @Override
    default void customize(QuerydslBindings bindings, QUser user) {

        bindings.bind(user.username).first((path, value) -> path.contains(value)) ❸
        bindings.bind(String.class)
            .first((StringPath path, String value) -> path.containsIgnoreCase(value)); ❹
        bindings.excluding(user.password); ❺
    }
}
```


- 1 `QuerydslPredicateExecutor` provides access to specific finder methods for `Predicate`.
- 2 `QuerydslBinderCustomizer` defined on the repository interface is automatically picked up and shortcuts `@QuerydslPredicate(bindings=...)`.
- 3 Define the binding for the `username` property to be a simple `contains` binding.
- 4 Define the default binding for `String` properties to be a case-insensitive `contains` match.
- 5 Exclude the `password` property from `Predicate` resolution.

4.8.3. Repository Populators

If you work with the Spring JDBC module, you are probably familiar with the support to populate a `DataSource` with SQL scripts. A similar abstraction is available on the repositories level, although it does not use SQL as the data definition language because it must be store-independent. Thus, the populators support XML (through Spring's OXM abstraction) and JSON (through Jackson) to define data with which to populate the repositories.

Assume you have a file `data.json` with the following content:

Example 47. Data defined in JSON

```
[ { "_class" : "com.acme.Person",
  "firstname" : "Dave",
  "lastname" : "Matthews" },
  { "_class" : "com.acme.Person",
  "firstname" : "Carter",
  "lastname" : "Beauford" } ]
```

You can populate your repositories by using the populator elements of the repository namespace provided in Spring Data Commons. To populate the preceding data to your `PersonRepository`, declare a populator similar to the following:

Example 48. Declaring a Jackson repository populator

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:repository="http://www.springframework.org/schema/data/repository"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/repository
    http://www.springframework.org/schema/data/repository/spring-repository.xsd">

  <repository:jackson2-populator locations="classpath:data.json" />

</beans>
```

The preceding declaration causes the `data.json` file to be read and deserialized by a Jackson `ObjectMapper`.

The type to which the JSON object is unmarshalled is determined by inspecting the `_class` attribute of the JSON document. The infrastructure eventually selects the appropriate repository to handle the object that was deserialized.

To instead use XML to define the data the repositories should be populated with, you can use the `unmarshaller-populator` element. You configure it to use one of the XML marshaller options available in Spring OXM. See the [Spring reference documentation](#) for details. The following example shows how to unmarshal a repository populator with JAXB:

Example 49. Declaring an unmarshalling repository populator (using JAXB)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:repository="http://www.springframework.org/schema/data/repository"
  xmlns:oxm="http://www.springframework.org/schema/oxm"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/repository
    http://www.springframework.org/schema/data/repository/spring-repository.xsd
    http://www.springframework.org/schema/oxm
    http://www.springframework.org/schema/oxm/spring-oxm.xsd">

  <repository:unmarshaller-populator locations="classpath:data.json"
    unmarshaller-ref="unmarshaller" />

  <oxm:jaxb2-marshaller contextPath="com.acme" />

</beans>
```

Reference Documentation

5. JPA Repositories

This chapter points out the specialties for repository support for JPA. This builds on the core repository support explained in “[Working with Spring Data Repositories](#)”. Make sure you have a sound understanding of the basic concepts explained there.

5.1. Introduction

This section describes the basics of configuring Spring Data JPA through either:

- “[Spring Namespace](#)” (XML configuration)
- “[Annotation-based Configuration](#)” (Java configuration)

5.1.1. Spring Namespace

The JPA module of Spring Data contains a custom namespace that allows defining repository beans. It also contains certain features and element attributes that are special to JPA. Generally, the JPA repositories can be set up by using the `repositories` element, as shown in the following example:

Example 50. Setting up JPA repositories by using the namespace

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jpa="http://www.springframework.org/schema/data/jpa"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/data/jpa
                           http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

    <jpa:repositories base-package="com.acme.repositories" />

</beans>
```


Using the `repositories` element looks up Spring Data repositories as described in “[Creating Repository Instances](#)”. Beyond that, it activates persistence exception translation for all beans annotated with `@Repository`, to let exceptions being thrown by the JPA persistence providers be converted into Spring’s `DataAccessException` hierarchy.

Custom Namespace Attributes

Beyond the default attributes of the `repositories` element, the JPA namespace offers additional attributes to let you gain more detailed control over the setup of the repositories:

Table 2. Custom JPA-specific attributes of the `repositories` element

entity-manager-factory-ref	Explicitly wire the <code>EntityManagerFactory</code> to be used with the repositories being detected by the <code>repositories</code> element. Usually used if multiple <code>EntityManagerFactory</code> beans are used within the application. If not configured, Spring Data automatically looks up the <code>EntityManagerFactory</code> bean with the name <code>entityManagerFactory</code> in the <code>ApplicationContext</code> .
transaction-manager-ref	Explicitly wire the <code>PlatformTransactionManager</code> to be used with the repositories being detected by the <code>repositories</code> element. Usually only necessary if multiple transaction managers or <code>EntityManagerFactory</code> beans have been configured. Default to a single defined <code>PlatformTransactionManager</code> inside the current <code>ApplicationContext</code> .



Spring Data JPA requires a `PlatformTransactionManager` bean named `transactionManager` to be present if no explicit `transaction-manager-ref` is defined.

5.1.2. Annotation-based Configuration

The Spring Data JPA repositories support can be activated not only through an XML namespace but also by using an annotation through JavaConfig, as shown in the following example:

Example 51. Spring Data JPA repositories using JavaConfig

```
@Configuration
@EnableJpaRepositories
```

```

@EnableTransactionManagement
class ApplicationConfig {

    @Bean
    public DataSource dataSource() {

        EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
        return builder.setType(EmbeddedDatabaseType.HSQL).build();
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory() {

        HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
        vendorAdapter.setGenerateDdl(true);

        LocalContainerEntityManagerFactoryBean factory = new LocalContainerEntityManagerFactoryBean();
        factory.setJpaVendorAdapter(vendorAdapter);
        factory.setPackagesToScan("com.acme.domain");
        factory.setDataSource(dataSource());
        return factory;
    }

    @Bean
    public PlatformTransactionManager transactionManager(EntityManagerFactory entityManagerFactory) {

        JpaTransactionManager txManager = new JpaTransactionManager();
        txManager.setEntityManagerFactory(entityManagerFactory);
        return txManager;
    }
}

```



You must create `LocalContainerEntityManagerFactoryBean` and not `EntityManagerFactory` directly, since the former also participates in exception translation mechanisms in addition to creating `EntityManagerFactory`.

The preceding configuration class sets up an embedded HSQL database by using the `EmbeddedDatabaseBuilder` API of `spring-jdbc`. Spring Data then sets up an `EntityManagerFactory` and uses Hibernate as the sample persistence provider. The last infrastructure component declared here is the `JpaTransactionManager`. Finally, the example activates Spring Data JPA repositories by using the `@EnableJpaRepositories` annotation, which essentially carries the same attributes as the XML namespace. If no base package is configured, it uses the one in which the configuration class resides.

5.1.3. Bootstrap Mode

By default, Spring Data JPA repositories are default Spring beans. They are singleton scoped and eagerly initialized. During startup, they already interact with the JPA `EntityManager` for verification and metadata analysis purposes. Spring Framework supports the initialization of the JPA `EntityManagerFactory` in a background thread because that process usually takes up a significant amount of startup time in a Spring application. To make use of that background initialization effectively, we need to make sure that JPA repositories are initialized as late as possible.

As of Spring Data JPA 2.1 you can now configure a `BootstrapMode` (either via the `@EnableJpaRepositories` annotation or the XML namespace) that takes the following values:

- **DEFAULT** (default) — Repositories are instantiated eagerly unless explicitly annotated with `@Lazy`. The lazification only has effect if no client bean needs an instance of the repository as that will require the initialization of the repository bean.
- **LAZY** — Implicitly declares all repository beans lazy and also causes lazy initialization proxies to be created to be injected into client beans. That means, that repositories will not get instantiated if the client bean is simply storing the instance in a field and not making use of the repository during initialization. Repository instances will be initialized and verified upon first interaction with the repository.
- **DEFERRED** — Fundamentally the same mode of operation as **LAZY**, but triggering repository initialization in response to an `ContextRefreshedEvent` so that repositories are verified before the application has completely started.

Recommendations

If you're not using asynchronous JPA bootstrap stick with the default bootstrap mode.

In case you bootstrap JPA asynchronously, **DEFERRED** is a reasonable default as it will make sure the Spring Data JPA bootstrap only waits for the `EntityManagerFactory` setup if that itself takes longer than initializing all other application components. Still, it makes sure that repositories are properly initialized and validated before the application signals it's up.

LAZY is a decent choice for testing scenarios and local development. Once you're pretty sure that repositories will properly bootstrap, or in cases where you're testing other parts of the application, executing verification for all repositories might just unnecessarily increase the startup time. The same applies to local development in which you only access parts of the application which might just need a single repository initialized.

5.2. Persisting Entities

This section describes how to persist (save) entities with Spring Data JPA.

5.2.1. Saving Entities

Saving an entity can be performed with the `CrudRepository.save(...)` method. It persists or merges the given entity by using the underlying JPA `EntityManager`. If the entity has not yet been persisted, Spring Data JPA saves the entity with a call to the `entityManager.persist(...)` method. Otherwise, it calls the `entityManager.merge(...)` method.

Entity State-detection Strategies

Spring Data JPA offers the following strategies to detect whether an entity is new or not:

- **Id-Property inspection (default)**: By default Spring Data JPA inspects the identifier property of the given entity. If the identifier property is `null`, then the entity is assumed to be new. Otherwise, it is assumed to be not new.
- **Implementing `Persistable`**: If an entity implements `Persistable`, Spring Data JPA delegates the new detection to the `isNew(...)` method of the entity. See the [JavaDoc](#) for details.
- **Implementing `EntityInformation`**: You can customize the `EntityInformation` abstraction used in the `SimpleJpaRepository` implementation by creating a subclass of `JpaRepositoryFactory` and overriding the `getEntityInformation(...)` method accordingly. You then have to register the custom

implementation of `JpaRepositoryFactory` as a Spring bean. Note that this should be rarely necessary. See the [JavaDoc](#) for details.

5.3. Query Methods

This section describes the various ways to create a query with Spring Data JPA.

5.3.1. Query Lookup Strategies

The JPA module supports defining a query manually as a String or having it being derived from the method name.

Declared Queries

Although getting a query derived from the method name is quite convenient, one might face the situation in which either the method name parser does not support the keyword one wants to use or the method name would get unnecessarily ugly. So you can either use JPA named queries through a naming convention (see [Using JPA Named Queries](#) for more information) or rather annotate your query method with `@Query` (see [Using @Query](#) for details).

5.3.2. Query Creation

Generally, the query creation mechanism for JPA works as described in “[Query methods](#)”. The following example shows what a JPA query method translates into:

Example 52. Query creation from method names

```
public interface UserRepository extends Repository<User, Long> {

    List<User> findByEmailAddressAndLastname(String emailAddress, String lastname);

}
```

We create a query using the JPA criteria API from this, but, essentially, this translates into the following query: `select u from User u where u.emailAddress = ?1 and u.lastname = ?2`. Spring Data JPA does a property check and traverses nested properties, as described in “[Property Expressions](#)”.

The following table describes the keywords supported for JPA and what a method containing that keyword translates to:

Table 3. Supported keywords inside method names

Keyword	Sample	JPQL snippet
And	<code>findByLastnameAndFirstname</code>	... where x.lastname = ?1 and x.firstname = ?2
Or	<code>findByLastnameOrFirstname</code>	... where x.lastname = ?1 or x.firstname = ?2

Keyword	Sample	JPQL snippet
Is, Equals	<code>findByFirstname</code> , <code>findByFirstnameIs</code> , <code>findByFirstnameEquals</code>	... where x.firstname = ?1
Between	<code>findByStartDateBetween</code>	... where x.startDate between ?1 and ?2
LessThan	<code>findByAgeLessThan</code>	... where x.age < ?1
LessThanEqual	<code>findByAgeLessThanEqual</code>	... where x.age <= ? 1
GreaterThan	<code>findByAgeGreaterThan</code>	... where x.age > ?1
GreaterThanEqual	<code>findByAgeGreaterThanEqual</code>	... where x.age >= ? 1
After	<code>findByStartDateAfter</code>	... where x.startDate > ?1
Before	<code>findByStartDateBefore</code>	... where x.startDate < ?1
IsNull	<code>findByAgeIsNull</code>	... where x.age is null
IsNotNull, NotNull	<code>findByAge(Is)NotNull</code>	... where x.age not null
Like	<code>findByFirstnameLike</code>	... where x.firstname like ?1
NotLike	<code>findByFirstnameNotLike</code>	... where x.firstname not like ?1
StartingWith	<code>findByFirstnameStartingWith</code>	... where x.firstname like ?1 (parameter bound with appended %)
EndingWith	<code>findByFirstnameEndingWith</code>	... where x.firstname like ?1 (parameter bound with prepended %)

Keyword	Sample	JPQL snippet
Containing	<code>findByFirstnameContaining</code>	<code>... where x.firstname like ?1 (parameter bound wrapped in %)</code>
OrderBy	<code>findByAgeOrderByLastnameDesc</code>	<code>... where x.age = ?1 order by x.lastname desc</code>
Not	<code>findByLastnameNot</code>	<code>... where x.lastname <> ?1</code>
In	<code>findByAgeIn(Collection<Age> ages)</code>	<code>... where x.age in ? 1</code>
NotIn	<code>findByAgeNotIn(Collection<Age> ages)</code>	<code>... where x.age not in ?1</code>
True	<code>findByActiveTrue()</code>	<code>... where x.active = true</code>
False	<code>findByActiveFalse()</code>	<code>... where x.active = false</code>
IgnoreCase	<code>findByFirstnameIgnoreCase</code>	<code>... where UPPER(x.firstname) = UPPER(?1)</code>



`In` and `NotIn` also take any subclass of `Collection` as a parameter as well as arrays or varargs. For other syntactical versions of the same logical operator, check “[Repository query keywords](#)”.

5.3.3. Using JPA Named Queries



The examples use the `<named-query />` element and `@NamedQuery` annotation. The queries for these configuration elements have to be defined in the JPA query language. Of course, you can use `<named-native-query />` or `@NamedNativeQuery` too. These elements let you define the query in native SQL by losing the database platform independence.

XML Named Query Definition

To use XML configuration, add the necessary `<named-query />` element to the `orm.xml` JPA configuration file located in the `META-INF` folder of your classpath. Automatic invocation of named queries is enabled by using some defined naming convention. For more details, see below.

Example 53. XML named query configuration

```
<named-query name="User.findByLastname">
  <query>select u from User u where u.lastname = ?1</query>
</named-query>
```

The query has a special name that is used to resolve it at runtime.

Annotation-based Configuration

Annotation-based configuration has the advantage of not needing another configuration file to be edited, lowering maintenance effort. You pay for that benefit by the need to recompile your domain class for every new query declaration.

Example 54. Annotation-based named query configuration

```
@Entity
@NamedQuery(name = "User.findByEmailAddress",
    query = "select u from User u where u.emailAddress = ?1")
public class User {
}
```

Declaring Interfaces

To allow execution of these named queries, specify the `UserRepository` as follows:

Example 55. Query method declaration in `UserRepository`

```
public interface UserRepository extends JpaRepository<User, Long> {

    List<User> findByLastname(String lastname);

    User findByEmailAddress(String emailAddress);
}
```

Spring Data tries to resolve a call to these methods to a named query, starting with the simple name of the configured domain class, followed by the method name separated by a dot. So the preceding example would use the named queries defined in the example instead of trying to create a query from the method name.

5.3.4. Using `@Query`

Using named queries to declare queries for entities is a valid approach and works fine for a small number of queries. As the queries themselves are tied to the Java method that executes them, you can actually bind them directly by using the Spring Data JPA `@Query` annotation rather than annotating them to the

domain class. This frees the domain class from persistence specific information and co-locates the query to the repository interface.

Queries annotated to the query method take precedence over queries defined using `@NamedQuery` or named queries declared in `orm.xml`.

The following example shows a query created with the `@Query` annotation:

Example 56. Declare query at the query method using `@Query`

```
public interface UserRepository extends JpaRepository<User, Long> {  
  
    @Query("select u from User u where u.emailAddress = ?1")  
    User findByEmailAddress(String emailAddress);  
}
```

Using Advanced LIKE Expressions

The query execution mechanism for manually defined queries created with `@Query` allows the definition of advanced LIKE expressions inside the query definition, as shown in the following example:

Example 57. Advanced like expressions in `@Query`

```
public interface UserRepository extends JpaRepository<User, Long> {  
  
    @Query("select u from User u where u.firstname like %?1")  
    List<User> findByFirstnameEndsWith(String firstname);  
}
```

In the preceding example, the LIKE delimiter character (%) is recognized, and the query is transformed into a valid JPQL query (removing the %). Upon query execution, the parameter passed to the method call gets augmented with the previously recognized LIKE pattern.

Native Queries

The `@Query` annotation allows for running native queries by setting the `nativeQuery` flag to true, as shown in the following example:

Example 58. Declare a native query at the query method using `@Query`

```
public interface UserRepository extends JpaRepository<User, Long> {  
  
    @Query(value = "SELECT * FROM USERS WHERE EMAIL_ADDRESS = ?1", nativeQuery = true)  
    User findByEmailAddress(String emailAddress);  
}
```



Spring Data JPA does not currently support dynamic sorting for native queries, because it would have to manipulate the actual query declared, which it cannot do reliably for

native SQL. You can, however, use native queries for pagination by specifying the count query yourself, as shown in the following example:

Example 59. Declare native count queries for pagination at the query method by using `@Query`

```
public interface UserRepository extends JpaRepository<User, Long> {

    @Query(value = "SELECT * FROM USERS WHERE LASTNAME = ?1",
        countQuery = "SELECT count(*) FROM USERS WHERE LASTNAME = ?1",
        nativeQuery = true)
    Page<User> findByLastname(String lastname, Pageable pageable);
}
```

A similar approach also works with named native queries, by adding the `.count` suffix to a copy of your query. You probably need to register a result set mapping for your count query, though.

5.3.5. Using Sort

Sorting can be done by either providing a `PageRequest` or by using `Sort` directly. The properties actually used within the `Order` instances of `Sort` need to match your domain model, which means they need to resolve to either a property or an alias used within the query. The JPQL defines this as a state field path expression.



Using any non-referenceable path expression leads to an `Exception`.

However, using `Sort` together with `@Query` lets you sneak in non-path-checked `Order` instances containing functions within the `ORDER BY` clause. This is possible because the `Order` is appended to the given query string. By default, Spring Data JPA rejects any `Order` instance containing function calls, but you can use `JpaSort.unsafe` to add potentially unsafe ordering.

The following example uses `Sort` and `JpaSort`, including an unsafe option on `JpaSort`:

Example 60. Using `Sort` and `JpaSort`

```
public interface UserRepository extends JpaRepository<User, Long> {

    @Query("select u from User u where u.lastname like ?1%")
    List<User> findByAndSort(String lastname, Sort sort);

    @Query("select u.id, LENGTH(u.firstname) as fn_len from User u where u.lastname like ?1%")
    List<Object[]> findByAsArrayAndSort(String lastname, Sort sort);
}

repo.findByAndSort("lannister", new Sort("firstname"));
repo.findByAndSort("stark", new Sort("LENGTH(firstname)"));
repo.findByAndSort("targaryen", JpaSort.unsafe("LENGTH(firstname)"));
repo.findByAsArrayAndSort("bolton", new Sort("fn_len"));
```

1
2
3
4

- 1 Valid `Sort` expression pointing to property in domain model.
- 2 Invalid `Sort` containing function call. Throws Exception.
- 3 Valid `Sort` containing explicitly `unsafe` Order.
- 4 Valid `Sort` expression pointing to aliased function.

5.3.6. Using Named Parameters

By default, Spring Data JPA uses position-based parameter binding, as described in all the preceding examples. This makes query methods a little error-prone when refactoring regarding the parameter position. To solve this issue, you can use `@Param` annotation to give a method parameter a concrete name and bind the name in the query, as shown in the following example:

Example 61. Using named parameters

```
public interface UserRepository extends JpaRepository<User, Long> {

    @Query("select u from User u where u.firstname = :firstname or u.lastname = :lastname")
    User findByLastnameOrFirstname(@Param("lastname") String lastname,
                                   @Param("firstname") String firstname);
}
```



The method parameters are switched according to their order in the defined query.



As of version 4, Spring fully supports Java 8's parameter name discovery based on the `-parameters` compiler flag. By using this flag in your build as an alternative to debug information, you can omit the `@Param` annotation for named parameters.

5.3.7. Using SpEL Expressions

As of Spring Data JPA release 1.4, we support the usage of restricted SpEL template expressions in manually defined queries that are defined with `@Query`. Upon query execution, these expressions are evaluated against a predefined set of variables. Spring Data JPA supports a variable called `entityName`. Its usage is `select x from #{entityName} x`. It inserts the `entityName` of the domain type associated with the given repository. The `entityName` is resolved as follows: If the domain type has set the name property on the `@Entity` annotation, it is used. Otherwise, the simple class-name of the domain type is used.

The following example demonstrates one use case for the `#{entityName}` expression in a query string where you want to define a repository interface with a query method and a manually defined query:

Example 62. Using SpEL expressions in repository query methods - `entityName`

```

@Entity
public class User {

    @Id
    @GeneratedValue
    Long id;

    String lastname;
}

public interface UserRepository extends JpaRepository<User, Long> {

    @Query("select u from #{#entityName} u where u.lastname = ?1")
    List<User> findByLastname(String lastname);
}

```

To avoid stating the actual entity name in the query string of a `@Query` annotation, you can use the `#{#entityName}` variable.



The `entityName` can be customized by using the `@Entity` annotation. Customizations in `orm.xml` are not supported for the SpEL expressions.

Of course, you could have just used `User` in the query declaration directly, but that would require you to change the query as well. The reference to `#{#entityName}` picks up potential future remappings of the `User` class to a different entity name (for example, by using `@Entity(name = "MyUser")`).

Another use case for the `#{#entityName}` expression in a query string is if you want to define a generic repository interface with specialized repository interfaces for a concrete domain type. To not repeat the definition of custom query methods on the concrete interfaces, you can use the entity name expression in the query string of the `@Query` annotation in the generic repository interface, as shown in the following example:

Example 63. Using SpEL expressions in repository query methods - entityName with inheritance

```

@MappedSuperclass
public abstract class AbstractMappedType {
    ...
    String attribute
}

@Entity
public class ConcreteType extends AbstractMappedType { ... }

@NoRepositoryBean
public interface MappedTypeRepository<T extends AbstractMappedType>
    extends Repository<T, Long> {

    @Query("select t from #{#entityName} t where t.attribute = ?1")
    List<T> findAllByAttribute(String attribute);
}

```

```
public interface ConcreteRepository
    extends MappedTypeRepository<ConcreteType> { ... }
```

In the preceding example, the `MappedTypeRepository` interface is the common parent interface for a few domain types extending `AbstractMappedType`. It also defines the generic `findAllByAttribute(...)` method, which can be used on instances of the specialized repository interfaces. If you now invoke `findAllByAttribute(...)` on `ConcreteRepository`, the query becomes `select t from ConcreteType t where t.attribute = ?1`.

5.3.8. Modifying Queries

All the previous sections describe how to declare queries to access a given entity or collection of entities. You can add custom modifying behavior by using the facilities described in “[Custom Implementations for Spring Data Repositories](#)”. As this approach is feasible for comprehensive custom functionality, you can modify queries that only need parameter binding by annotating the query method with `@Modifying`, as shown in the following example:

Example 64. Declaring manipulating queries

```
@Modifying
@Query("update User u set u.firstname = ?1 where u.lastname = ?2")
int setFixedFirstnameFor(String firstname, String lastname);
```

Doing so triggers the query annotated to the method as an updating query instead of a selecting one. As the `EntityManager` might contain outdated entities after the execution of the modifying query, we do not automatically clear it (see the [JavaDoc](#) of `EntityManager.clear()` for details), since this effectively drops all non-flushed changes still pending in the `EntityManager`. If you wish the `EntityManager` to be cleared automatically, you can set the `@Modifying` annotation’s `clearAutomatically` attribute to `true`.

Derived Delete Queries

Spring Data JPA also supports derived delete queries that let you avoid having to declare the JPQL query explicitly, as shown in the following example:

Example 65. Using a derived delete query

```
interface UserRepository extends Repository<User, Long> {

    void deleteByRoleId(long roleId);

    @Modifying
    @Query("delete from User u where user.role.id = ?1")
    void deleteInBulkByRoleId(long roleId);
}
```

Although the `deleteByRoleId(...)` method looks like it basically produces the same result as the `deleteInBulkByRoleId(...)`, there is an important difference between the two method declarations in terms of the way they get executed. As the name suggests, the latter method issues a single JPQL query

(the one defined in the annotation) against the database. This means even currently loaded instances of `User` do not see lifecycle callbacks invoked.

To make sure lifecycle queries are actually invoked, an invocation of `deleteByRoleId(...)` executes a query and then deletes the returned instances one by one, so that the persistence provider can actually invoke `@PreRemove` callbacks on those entities.

In fact, a derived delete query is a shortcut for executing the query and then calling `CrudRepository.delete(Iterable<User> users)` on the result and keeping behavior in sync with the implementations of other `delete(...)` methods in `CrudRepository`.

5.3.9. Applying Query Hints

To apply JPA query hints to the queries declared in your repository interface, you can use the `@QueryHints` annotation. It takes an array of JPA `@QueryHint` annotations plus a boolean flag to potentially disable the hints applied to the additional count query triggered when applying pagination, as shown in the following example:

Example 66. Using QueryHints with a repository method

```
public interface UserRepository extends Repository<User, Long> {

    @QueryHints(value = { @QueryHint(name = "name", value = "value")},
                  forCounting = false)
    Page<User> findByLastname(String lastname, Pageable pageable);

}
```

The preceding declaration would apply the configured `@QueryHint` for that actually query but omit applying it to the count query triggered to calculate the total number of pages.

5.3.10. Configuring Fetch- and LoadGraphs

The JPA 2.1 specification introduced support for specifying Fetch- and LoadGraphs that we also support with the `@EntityGraph` annotation, which lets you reference a `@NamedEntityGraph` definition. You can use that annotation on an entity to configure the fetch plan of the resulting query. The type (Fetch or Load) of the fetching can be configured by using the `type` attribute on the `@EntityGraph` annotation. See the JPA 2.1 Spec 3.7.4 for further reference.

The following example shows how to define a named entity graph on an entity:

Example 67. Defining a named entity graph on an entity.

```
@Entity
@NamedEntityGraph(name = "GroupInfo.detail",
                  attributeNodes = @NamedAttributeNode("members"))
public class GroupInfo {

    // default fetch mode is lazy.
    @ManyToMany
    List<GroupMember> members = new ArrayList<GroupMember>();

    ...
}
```

The following example shows how to reference a named entity graph on a repository query method:

Example 68. Referencing a named entity graph definition on a repository query method.

```
@Repository
public interface GroupRepository extends CrudRepository<GroupInfo, String> {

    @EntityGraph(value = "GroupInfo.detail", type = EntityGraphType.LOAD)
    GroupInfo getByName(String name);

}
```

It is also possible to define ad hoc entity graphs by using `@EntityGraph`. The provided `attributePaths` are translated into the according `EntityGraph` without needing to explicitly add `@NamedEntityGraph` to your domain types, as shown in the following example:

Example 69. Using AD-HOC entity graph definition on an repository query method.

```
@Repository
public interface GroupRepository extends CrudRepository<GroupInfo, String> {

    @EntityGraph(attributePaths = { "members" })
    GroupInfo getByName(String name);

}
```

5.3.11. Projections

Spring Data query methods usually return one or multiple instances of the aggregate root managed by the repository. However, it might sometimes be desirable to create projections based on certain attributes of those types. Spring Data allows modeling dedicated return types, to more selectively retrieve partial views of the managed aggregates.

Imagine a repository and aggregate root type such as the following example:

Example 70. A sample aggregate and repository

```
class Person {

    @Id UUID id;
    String firstname, lastname;
    Address address;

    static class Address {
        String zipCode, city, street;
    }
}

interface PersonRepository extends Repository<Person, UUID> {

    Collection<Person> findByLastname(String lastname);

}
```


Now imagine that we want to retrieve the person's name attributes only. What means does Spring Data offer to achieve this? The rest of this chapter answers that question.

Interface-based Projections

The easiest way to limit the result of the queries to only the name attributes is by declaring an interface that exposes accessor methods for the properties to be read, as shown in the following example:

Example 71. A projection interface to retrieve a subset of attributes

```
interface NamesOnly {  
  
    String getFirstname();  
    String getLastName();  
}
```

The important bit here is that the properties defined here exactly match properties in the aggregate root. Doing so lets a query method be added as follows:

Example 72. A repository using an interface based projection with a query method

```
interface PersonRepository extends Repository<Person, UUID> {  
  
    Collection<NamesOnly> findByLastname(String lastname);  
}
```

The query execution engine creates proxy instances of that interface at runtime for each element returned and forwards calls to the exposed methods to the target object.

Projections can be used recursively. If you want to include some of the `Address` information as well, create a projection interface for that and return that interface from the declaration of `getAddress()`, as shown in the following example:

Example 73. A projection interface to retrieve a subset of attributes

```
interface PersonSummary {  
  
    String getFirstname();  
    String getLastName();  
    AddressSummary getAddress();  
  
    interface AddressSummary {  
        String getCity();  
    }  
}
```

On method invocation, the `address` property of the target instance is obtained and wrapped into a projecting proxy in turn.

Closed Projections

A projection interface whose accessor methods all match properties of the target aggregate is considered to be a closed projection. The following example (which we used earlier in this chapter, too) is a closed projection:

Example 74. A closed projection

```
interface NamesOnly {  
  
    String getFirstname();  
    String getLastName();  
}
```

If you use a closed projection, Spring Data can optimize the query execution, because we know about all the attributes that are needed to back the projection proxy. For more details on that, see the module-specific part of the reference documentation.

Open Projections

Accessor methods in projection interfaces can also be used to compute new values by using the `@Value` annotation, as shown in the following example:

Example 75. An Open Projection

```
interface NamesOnly {  
  
    @Value("#{target.firstname + ' ' + target.lastname}")  
    String getFullName();  
    ...  
}
```

The aggregate root backing the projection is available in the `target` variable. A projection interface using `@Value` is an open projection. Spring Data cannot apply query execution optimizations in this case, because the SpEL expression could use any attribute of the aggregate root.

The expressions used in `@Value` should not be too complex — you want to avoid programming in `String` variables. For very simple expressions, one option might be to resort to default methods (introduced in Java 8), as shown in the following example:

Example 76. A projection interface using a default method for custom logic

```
interface NamesOnly {  
  
    String getFirstname();  
    String getLastName();  
  
    default String getFullName() {  
        return getFirstname().concat(" ").concat(getLastName());  
    }  
}
```

This approach requires you to be able to implement logic purely based on the other accessor methods exposed on the projection interface. A second, more flexible, option is to implement the custom logic in a Spring bean and then invoke that from the SpEL expression, as shown in the following example:

Example 77. Sample Person object

```
@Component
class MyBean {

    String getFullName(Person person) {
        ...
    }
}

interface NamesOnly {

    @Value("#{@myBean.getFullName(target)}")
    String getFullName();
    ...
}
```

Notice how the SpEL expression refers to `myBean` and invokes the `getFullName(...)` method and forwards the projection target as a method parameter. Methods backed by SpEL expression evaluation can also use method parameters, which can then be referred to from the expression. The method parameters are available through an `Object` array named `args`. The following example shows how to get a method parameter from the `args` array:

Example 78. Sample Person object

```
interface NamesOnly {

    @Value("#{args[0] + ' ' + target.firstname + '!'}")
    String getSalutation(String prefix);
}
```

Again, for more complex expressions, you should use a Spring bean and let the expression invoke a method, as described [earlier](#).

Class-based Projections (DTOs)

Another way of defining projections is by using value type DTOs (Data Transfer Objects) that hold properties for the fields that are supposed to be retrieved. These DTO types can be used in exactly the same way projection interfaces are used, except that no proxying happens and no nested projections can be applied.

If the store optimizes the query execution by limiting the fields to be loaded, the fields to be loaded are determined from the parameter names of the constructor that is exposed.

The following example shows a projecting DTO:

Example 79. A projecting DTO

```

class NamesOnly {

    private final String firstname, lastname;

    NamesOnly(String firstname, String lastname) {

        this.firstname = firstname;
        this.lastname = lastname;
    }

    String getFirstname() {
        return this.firstname;
    }

    String getLastname() {
        return this.lastname;
    }

    // equals(...) and hashCode() implementations
}

```

Avoid boilerplate code for projection DTOs

You can dramatically simplify the code for a DTO by using [Project Lombok](#), which provides an `@Value` annotation (not to be confused with Spring's `@Value` annotation shown in the earlier interface examples). If you use Project Lombok's `@Value` annotation, the sample DTO shown earlier would become the following:



```

@Value
class NamesOnly {
    String firstname, lastname;
}

```

Fields are `private final` by default, and the class exposes a constructor that takes all fields and automatically gets `equals(...)` and `hashCode()` methods implemented.

Dynamic Projections

So far, we have used the projection type as the return type or element type of a collection. However, you might want to select the type to be used at invocation time (which makes it dynamic). To apply dynamic projections, use a query method such as the one shown in the following example:

Example 80. A repository using a dynamic projection parameter

```

interface PersonRepository extends Repository<Person, UUID> {

    <T> Collection<T> findByLastname(String lastname, Class<T> type);
}

```

This way, the method can be used to obtain the aggregates as is or with a projection applied, as shown in the following example:

Example 81. Using a repository with dynamic projections

```
void someMethod(PersonRepository people) {  
  
    Collection<Person> aggregates =  
        people.findByLastname("Matthews", Person.class);  
  
    Collection<NamesOnly> aggregates =  
        people.findByLastname("Matthews", NamesOnly.class);  
}
```

5.4. Stored Procedures

The JPA 2.1 specification introduced support for calling stored procedures by using the JPA criteria query API. We introduced the `@Procedure` annotation for declaring stored procedure metadata on a repository method.

The examples to follow use the following procedure:

Example 82. The definition of the `plus1inout` procedure in HSQL DB.

```
/*;  
DROP procedure IF EXISTS plus1inout  
/*;  
CREATE procedure plus1inout (IN arg int, OUT res int)  
BEGIN ATOMIC  
    set res = arg + 1;  
END  
/*;
```

Metadata for stored procedures can be configured by using the `NamedStoredProcedureQuery` annotation on an entity type.

Example 83. StoredProcedure metadata definitions on an entity.

```
@Entity  
@NamedStoredProcedureQuery(name = "User.plus1", procedureName = "plus1inout", parameters = {  
    @StoredProcedureParameter(mode = ParameterMode.IN, name = "arg", type = Integer.class),  
    @StoredProcedureParameter(mode = ParameterMode.OUT, name = "res", type = Integer.class) })  
public class User {}
```

You can reference stored procedures from a repository method in multiple ways. The stored procedure to be called can either be defined directly by using the `value` or `procedureName` attribute of the `@Procedure` annotation or indirectly by using the `name` attribute. If no name is configured, the name of the repository method is used as a fallback.

The following example shows how to reference an explicitly mapped procedure:

Example 84. Referencing explicitly mapped procedure with name "plus1inout" in database.

```
@Procedure("plus1inout")
Integer explicitlyNamedPlus1inout(Integer arg);
```

The following example shows how to reference an implicitly mapped procedure by using a `procedureName` alias:

Example 85. Referencing implicitly mapped procedure with name "plus1inout" in database via `procedureName` alias.

```
@Procedure(procedureName = "plus1inout")
Integer plus1inout(Integer arg);
```

The following example shows how to reference an explicitly mapped named procedure in `EntityManager`:

Example 86. Referencing explicitly mapped named stored procedure "User.plus1IO" in `EntityManager`.

```
@Procedure(name = "User.plus1IO")
Integer entityAnnotatedCustomNamedProcedurePlus1IO(@Param("arg") Integer arg);
```

The following example shows how to reference an implicitly named stored procedure in `EntityManager` by using the method name:

Example 87. Referencing implicitly mapped named stored procedure "User.plus1" in `EntityManager` by using the method name.

```
@Procedure
Integer plus1(@Param("arg") Integer arg);
```

5.5. Specifications

JPA 2 introduces a criteria API that you can use to build queries programmatically. By writing a `criteria`, you define the where clause of a query for a domain class. Taking another step back, these criteria can be regarded as a predicate over the entity that is described by the JPA criteria API constraints.

Spring Data JPA takes the concept of a specification from Eric Evans' book, "Domain Driven Design", following the same semantics and providing an API to define such specifications with the JPA criteria API. To support specifications, you can extend your repository interface with the `JpaSpecificationExecutor` interface, as follows:

```
public interface CustomerRepository extends CrudRepository<Customer, Long>, JpaSpecificationExecutor {
    ...
}
```

The additional interface has methods that let you execute specifications in a variety of ways. For example, the `findAll` method returns all entities that match the specification, as shown in the following example:

```
List<T> findAll(Specification<T> spec);
```

The `Specification` interface is defined as follows:

```
public interface Specification<T> {
    Predicate toPredicate(Root<T> root, CriteriaQuery<?> query,
        CriteriaBuilder builder);
}
```

Specifications can easily be used to build an extensible set of predicates on top of an entity that then can be combined and used with `JpaRepository` without the need to declare a query (method) for every needed combination, as shown in the following example:

Example 88. Specifications for a Customer

```
public class CustomerSpecs {

    public static Specification<Customer> isLongTermCustomer() {
        return new Specification<Customer>() {
            public Predicate toPredicate(Root<Customer> root, CriteriaQuery<?> query,
                CriteriaBuilder builder) {

                LocalDate date = new LocalDate().minusYears(2);
                return builder.lessThan(root.get(_Customer.createdAt), date);
            }
        };
    }

    public static Specification<Customer> hasSalesOfMoreThan(MontaryAmount value) {
        return new Specification<Customer>() {
            public Predicate toPredicate(Root<T> root, CriteriaQuery<?> query,
                CriteriaBuilder builder) {

                // build query here
            }
        };
    }
}
```

Admittedly, the amount of boilerplate leaves room for improvement (that may eventually be reduced by Java 8 closures), but the client side becomes much nicer, as you will see later in this section. The `_Customer` type is a metamodel type generated using the JPA Metamodel generator (see the [Hibernate implementation's documentation for an example](#)). So the expression, `_Customer.createdAt`, assumes the `Customer` has a `createdAt` attribute of type `Date`. Besides that, we have expressed some criteria on a business requirement abstraction level and created executable `Specifications`. So a client might use a `Specification` as follows:

Example 89. Using a simple Specification

```
List<Customer> customers = customerRepository.findAll(isLongTermCustomer());
```

Why not create a query for this kind of data access? Using a single `Specification` does not gain a lot of benefit over a plain query declaration. The power of specifications really shines when you combine them to create new `Specification` objects. You can achieve this through the `Specifications` helper class we provide to build expressions similar to the following:

Example 90. Combined Specifications

```
MonetaryAmount amount = new MonetaryAmount(200.0, Currencies.DOLLAR);
List<Customer> customers = customerRepository.findAll(
    where(isLongTermCustomer()).or(hasSalesOfMoreThan(amount)));
```

`Specifications` offers some “glue-code” methods to chain and combine `Specification` instances. These methods let you extend your data access layer by creating new `Specification` implementations and combining them with already existing implementations.

5.6. Query by Example

5.6.1. Introduction

This chapter provides an introduction to Query by Example and explains how to use it.

Query by Example (QBE) is a user-friendly querying technique with a simple interface. It allows dynamic query creation and does not require you to write queries that contain field names. In fact, Query by Example does not require you to write queries by using store-specific query languages at all.

5.6.2. Usage

The Query by Example API consists of three parts:

- **Probe**: The actual example of a domain object with populated fields.
- **ExampleMatcher**: The `ExampleMatcher` carries details on how to match particular fields. It can be reused across multiple Examples.
- **Example**: An `Example` consists of the probe and the `ExampleMatcher`. It is used to create the query.

Query by Example is well suited for several use cases:

- Querying your data store with a set of static or dynamic constraints.
- Frequent refactoring of the domain objects without worrying about breaking existing queries.
- Working independently from the underlying data store API.

Query by Example also has several limitations:

- No support for nested or grouped property constraints, such as `firstname = ?0 or (firstname = ?1 and lastname = ?2)`.
- Only supports starts/contains/ends/regex matching for strings and exact matching for other property types.

Before getting started with Query by Example, you need to have a domain object. To get started, create an interface for your repository, as shown in the following example:

Example 91. Sample Person object

```
public class Person {  
  
    @Id  
    private String id;  
    private String firstname;  
    private String lastname;  
    private Address address;  
  
    // ... getters and setters omitted  
}
```

The preceding example shows a simple domain object. You can use it to create an `Example`. By default, fields having `null` values are ignored, and strings are matched by using the store specific defaults. Examples can be built by either using the `of` factory method or by using `ExampleMatcher`. `Example` is immutable. The following listing shows a simple `Example`:

Example 92. Simple Example

```
Person person = new Person();  
person.setFirstname("Dave");  
  
Example<Person> example = Example.of(person);
```

- ❶ Create a new instance of the domain object.
- ❷ Set the properties to query.
- ❸ Create the `Example`.

Examples are ideally be executed with repositories. To do so, let your repository interface extend `QueryByExampleExecutor<T>`. The following listing shows an excerpt from the `QueryByExampleExecutor` interface:

Example 93. The QueryByExampleExecutor

```
public interface QueryByExampleExecutor<T> {  
  
    <S extends T> S findOne(Example<S> example);  
  
    <S extends T> Iterable<S> findAll(Example<S> example);  
  
    // ... more functionality omitted.  
}
```

5.6.3. Example Matchers

Examples are not limited to default settings. You can specify your own defaults for string matching, null handling, and property-specific settings by using the `ExampleMatcher`, as shown in the following example:

Example 94. Example matcher with customized matching

```
Person person = new Person();           1
person.setFirstname("Dave");             2

ExampleMatcher matcher = ExampleMatcher.matching()  3
    .withIgnorePaths("lastname")                4
    .withIncludeNullValues()                     5
    .withStringMatcherEnding();                  6

Example<Person> example = Example.of(person, matcher); 7
```

- 1 Create a new instance of the domain object.
- 2 Set properties.
- 3 Create an `ExampleMatcher` to expect all values to match. It is usable at this stage even without further configuration.
- 4 Construct a new `ExampleMatcher` to ignore the `lastname` property path.
- 5 Construct a new `ExampleMatcher` to ignore the `lastname` property path and to include null values.
- 6 Construct a new `ExampleMatcher` to ignore the `lastname` property path, to include null values, and to perform suffix string matching.
- 7 Create a new `Example` based on the domain object and the configured `ExampleMatcher`.

By default, the `ExampleMatcher` expects all values set on the probe to match. If you want to get results matching any of the predicates defined implicitly, use `ExampleMatcher.matchingAny()`.

You can specify behavior for individual properties (such as "firstname" and "lastname" or, for nested properties, "address.city"). You can tune it with matching options and case sensitivity, as shown in the following example:

Example 95. Configuring matcher options

```
ExampleMatcher matcher = ExampleMatcher.matching()
    .withMatcher("firstname", endsWith())
    .withMatcher("lastname", startsWith().ignoreCase());
}
```

Another way to configure matcher options is to use lambdas (introduced in Java 8). This approach creates a callback that asks the implementor to modify the matcher. You need not return the matcher, because configuration options are held within the matcher instance. The following example shows a matcher that uses lambdas:

Example 96. Configuring matcher options with lambdas

```
ExampleMatcher matcher = ExampleMatcher.matching()
    .withMatcher("firstname", match -> match.endsWith())
    .withMatcher("firstname", match -> match.startsWith());
}
```

Queries created by `Example` use a merged view of the configuration. Default matching settings can be set at the `ExampleMatcher` level, while individual settings can be applied to particular property paths. Settings that are set on `ExampleMatcher` are inherited by property path settings unless they are defined explicitly. Settings on a property patch have higher precedence than default settings. The following table describes the scope of the various `ExampleMatcher` settings:

Table 4. Scope of `ExampleMatcher` settings

Setting	Scope
Null-handling	<code>ExampleMatcher</code>
String matching	<code>ExampleMatcher</code> and property path
Ignoring properties	Property path
Case sensitivity	<code>ExampleMatcher</code> and property path
Value transformation	Property path

5.6.4. Executing an example

In Spring Data JPA, you can use Query by Example with Repositories, as shown in the following example:

Example 97. Query by Example using a Repository

```
public interface PersonRepository extends JpaRepository<Person, String> { ... }

public class PersonService {

    @Autowired PersonRepository personRepository;

    public List<Person> findPeople(Person probe) {
        return personRepository.findAll(Example.of(probe));
    }
}
```



Currently, only `SingularAttribute` properties can be used for property matching.

The property specifier accepts property names (such as `firstname` and `lastname`). You can navigate by chaining properties together with dots (`address.city`). You can also tune it with matching options and case sensitivity.

The following table shows the various `StringMatcher` options that you can use and the result of using them on a field named `firstname`:

Table 5. StringMatcher options

Matching	Logical result
DEFAULT (case-sensitive)	<code>firstname = ?0</code>
DEFAULT (case-insensitive)	<code>LOWER(firstname) = LOWER(?0)</code>
EXACT (case-sensitive)	<code>firstname = ?0</code>
EXACT (case-insensitive)	<code>LOWER(firstname) = LOWER(?0)</code>
STARTING (case-sensitive)	<code>firstname like ?0 + '%'</code>
STARTING (case-insensitive)	<code>LOWER(firstname) like LOWER(?0) + '%'</code>
ENDING (case-sensitive)	<code>firstname like '%' + ?0</code>
ENDING (case-insensitive)	<code>LOWER(firstname) like '%' + LOWER(?0)</code>
CONTAINING (case-sensitive)	<code>firstname like '%' + ?0 + '%'</code>
CONTAINING (case-insensitive)	<code>LOWER(firstname) like '%' + LOWER(?0) + '%'</code>

5.7. Transactionality

By default, CRUD methods on repository instances are transactional. For read operations, the transaction configuration `readOnly` flag is set to `true`. All others are configured with a plain `@Transactional` so that default transaction configuration applies. For details, see JavaDoc of [SimpleJpaRepository](#). If you need to tweak transaction configuration for one of the methods declared in a repository, redeclare the method in your repository interface, as follows:

Example 98. Custom transaction configuration for CRUD

```
public interface UserRepository extends CrudRepository<User, Long> {

    @Override
    @Transactional(timeout = 10)
    public List<User> findAll();

    // Further query method declarations
}
```

Doing so causes the `findAll()` method to run with a timeout of 10 seconds and without the `readOnly` flag.

Another way to alter transactional behaviour is to use a facade or service implementation that (typically) covers more than one repository. Its purpose is to define transactional boundaries for non-CRUD operations. The following example shows how to use such a facade for more than one repository:

Example 99. Using a facade to define transactions for multiple repository calls

```

@Service
class UserManagementImpl implements UserManagement {

    private final UserRepository userRepository;
    private final RoleRepository roleRepository;

    @Autowired
    public UserManagementImpl(UserRepository userRepository,
        RoleRepository roleRepository) {
        this.userRepository = userRepository;
        this.roleRepository = roleRepository;
    }

    @Transactional
    public void addRoleToAllUsers(String roleName) {

        Role role = roleRepository.findByName(roleName);

        for (User user : userRepository.findAll()) {
            user.addRole(role);
            userRepository.save(user);
        }
    }
}

```

This example causes call to `addRoleToAllUsers(...)` to run inside a transaction (participating in an existing one or creating a new one if none are already running). The transaction configuration at the repositories is then neglected, as the outer transaction configuration determines the actual one used. Note that you must activate `<tx:annotation-driven />` or use `@EnableTransactionManagement` explicitly to get annotation-based configuration of facades to work. This example assumes you use component scanning.

5.7.1. Transactional query methods

To let your query methods be transactional, use `@Transactional` at the repository interface you define, as shown in the following example:

Example 100. Using @Transactional at query methods

```

@Transactional(readOnly = true)
public interface UserRepository extends JpaRepository<User, Long> {

    List<User> findByLastname(String lastname);

    @Modifying
    @Transactional
    @Query("delete from User u where u.active = false")
    void deleteInactiveUsers();
}

```

Typically, you want the `readOnly` flag to be set to `true`, as most of the query methods only read data. In contrast to that, `deleteInactiveUsers()` makes use of the `@Modifying` annotation and overrides the transaction configuration. Thus, the method runs with the `readOnly` flag set to `false`.



You can use transactions for read-only queries and mark them as such by setting the `readOnly` flag. Doing so does not, however, act as a check that you do not trigger a manipulating query (although some databases reject `INSERT` and `UPDATE` statements inside a read-only transaction). The `readOnly` flag is instead propagated as a hint to the underlying JDBC driver for performance optimizations. Furthermore, Spring performs some optimizations on the underlying JPA provider. For example, when used with Hibernate, the flush mode is set to `NEVER` when you configure a transaction as `readOnly`, which causes Hibernate to skip dirty checks (a noticeable improvement on large object trees).

5.8. Locking

To specify the lock mode to be used, you can use the `@Lock` annotation on query methods, as shown in the following example:

Example 101. Defining lock metadata on query methods

```
interface UserRepository extends Repository<User, Long> {  
  
    // Plain query method  
    @Lock(LockModeType.READ)  
    List<User> findByLastname(String lastname);  
}
```

This method declaration causes the query being triggered to be equipped with a `LockModeType` of `READ`. You can also define locking for CRUD methods by redeclaring them in your repository interface and adding the `@Lock` annotation, as shown in the following example:

Example 102. Defining lock metadata on CRUD methods

```
interface UserRepository extends Repository<User, Long> {  
  
    // Redclaration of a CRUD method  
    @Lock(LockModeType.READ);  
    List<User> findAll();  
}
```

5.9. Auditing

5.9.1. Basics

Spring Data provides sophisticated support to transparently keep track of who created or changed an entity and when the change happened. To benefit from that functionality, you have to equip your entity classes with auditing metadata that can be defined either using annotations or by implementing an interface.

Annotation-based Auditing Metadata

We provide `@CreatedBy` and `@LastModifiedBy` to capture the user who created or modified the entity as well as `@CreatedDate` and `@LastModifiedDate` to capture when the change happened.

Example 103. An audited entity

```
class Customer {  
  
    @CreatedBy  
    private User user;  
  
    @CreatedDate  
    private DateTime createdDate;  
  
    // ... further properties omitted  
}
```

As you can see, the annotations can be applied selectively, depending on which information you want to capture. The annotations capturing when changes were made can be used on properties of type `Joda-Time`, `DateTime`, legacy Java `Date` and `Calendar`, JDK8 date and time types, and `long` or `Long`.

Interface-based Auditing Metadata

In case you do not want to use annotations to define auditing metadata, you can let your domain class implement the `Auditable` interface. It exposes setter methods for all of the auditing properties.

There is also a convenience base class, `AbstractAuditable`, which you can extend to avoid the need to manually implement the interface methods. Doing so increases the coupling of your domain classes to Spring Data, which might be something you want to avoid. Usually, the annotation-based way of defining auditing metadata is preferred as it is less invasive and more flexible.

AuditorAware

In case you use either `@CreatedBy` or `@LastModifiedBy`, the auditing infrastructure somehow needs to become aware of the current principal. To do so, we provide an `AuditorAware<T>` SPI interface that you have to implement to tell the infrastructure who the current user or system interacting with the application is. The generic type `T` defines what type the properties annotated with `@CreatedBy` or `@LastModifiedBy` have to be.

The following example shows an implementation of the interface that uses Spring Security's `Authentication` object:

Example 104. Implementation of AuditorAware based on Spring Security

```
class SpringSecurityAuditorAware implements AuditorAware<User> {  
  
    public User getCurrentAuditor() {  
  
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();  
  
        if (authentication == null || !authentication.isAuthenticated()) {  
            return null;  
        }  
  
        return ((MyUserDetails) authentication.getPrincipal()).getUser();  
    }  
}
```

```
}
}
```

The implementation accesses the `Authentication` object provided by Spring Security and looks up the custom `UserDetails` instance that you have created in your `UserDetailsService` implementation. We assume here that you are exposing the domain user through the `UserDetails` implementation but that, based on the `Authentication` found, you could also look it up from anywhere. `:leveloffset: -1`

5.9.2. JPA Auditing

General Auditing Configuration

Spring Data JPA ships with an entity listener that can be used to trigger the capturing of auditing information. First, you must register the `AuditingEntityListener` to be used for all entities in your persistence contexts inside your `orm.xml` file, as shown in the following example:

Example 105. Auditing configuration `orm.xml`

```
<persistence-unit-metadata>
  <persistence-unit-defaults>
    <entity-listeners>
      <entity-listener class="...data.jpa.domain.support.AuditingEntityListener" />
    </entity-listeners>
  </persistence-unit-defaults>
</persistence-unit-metadata>
```

You can also enable the `AuditingEntityListener` on a per-entity basis by using the `@EntityListeners` annotation, as follows:

```
@Entity
@EntityListeners(AuditingEntityListener.class)
public class MyEntity {

}
```



The auditing feature requires `spring-aspects.jar` to be on the classpath.

With `orm.xml` suitably modified and `spring-aspects.jar` on the classpath, activating auditing functionality is a matter of adding the Spring Data JPA `auditing` namespace element to your configuration, as follows:

Example 106. Activating auditing using XML configuration

```
<jpa:auditing auditor-aware-ref="yourAuditorAwareBean" />
```


As of Spring Data JPA 1.5, you can enable auditing by annotating a configuration class with the `@EnableJpaAuditing` annotation. You must still modify the `orm.xml` file and have `spring-aspects.jar` on the classpath. The following example shows how to use the `@EnableJpaAuditing` annotation:

Example 107. Activating auditing with Java configuration

```
@Configuration
@EnableJpaAuditing
class Config {

    @Bean
    public AuditorAware<AuditableUser> auditorProvider() {
        return new AuditorAwareImpl();
    }
}
```

If you expose a bean of type `AuditorAware` to the `ApplicationContext`, the auditing infrastructure automatically picks it up and uses it to determine the current user to be set on domain types. If you have multiple implementations registered in the `ApplicationContext`, you can select the one to be used by explicitly setting the `auditorAwareRef` attribute of `@EnableJpaAuditing`.

5.10. Miscellaneous Considerations

5.10.1. Using `JpaContext` in Custom Implementations

When working with multiple `EntityManager` instances and [custom repository implementations](#), you need to wire the correct `EntityManager` into the repository implementation class. You can do so by explicitly naming the `EntityManager` in the `@PersistenceContext` annotation or, if the `EntityManager` is `@Autowired`, by using `@Qualifier`.

As of Spring Data JPA 1.9, Spring Data JPA includes a class called `JpaContext` that lets you obtain the `EntityManager` by managed domain class, assuming it is managed by only one of the `EntityManager` instances in the application. The following example shows how to use `JpaContext` in a custom repository:

Example 108. Using `JpaContext` in a custom repository implementation

```
class UserRepositoryImpl implements UserRepositoryCustom {

    private final EntityManager em;

    @Autowired
    public UserRepositoryImpl(JpaContext context) {
        this.em = context.getEntityManagerByManagedType(User.class);
    }

    ...
}
```

The advantage of this approach is that, if the domain type gets assigned to a different persistence unit, the repository does not have to be touched to alter the reference to the persistence unit.

5.10.2. Merging persistence units

Spring supports having multiple persistence units. Sometimes, however, you might want to modularize your application but still make sure that all these modules run inside a single persistence unit. To enable that behavior, Spring Data JPA offers a `PersistenceUnitManager` implementation that automatically merges persistence units based on their name, as shown in the following example:

Example 109. Using `MergingPersistenceUnitManager`

```
<bean class="...LocalContainerEntityManagerFactoryBean">
  <property name="persistenceUnitManager">
    <bean class="...MergingPersistenceUnitManager" />
  </property>
</bean>
```

Classpath Scanning for @Entity Classes and JPA Mapping Files

A plain JPA setup requires all annotation-mapped entity classes to be listed in `orm.xml`. The same applies to XML mapping files. Spring Data JPA provides a `ClasspathScanningPersistenceUnitPostProcessor` that gets a base package configured and optionally takes a mapping filename pattern. It then scans the given package for classes annotated with `@Entity` or `@MappedSuperclass`, loads the configuration files that match the filename pattern, and hands them to the JPA configuration. The post-processor must be configured as follows:

Example 110. Using `ClasspathScanningPersistenceUnitPostProcessor`

```
<bean class="...LocalContainerEntityManagerFactoryBean">
  <property name="persistenceUnitPostProcessors">
    <list>
      <bean
class="org.springframework.data.jpa.support.ClasspathScanningPersistenceUnitPostProcessor">
        <constructor-arg value="com.acme.domain" />
        <property name="mappingFileNamePattern" value="**/*Mapping.xml" />
      </bean>
    </list>
  </property>
</bean>
```



As of Spring 3.1, a package to scan can be configured on the `LocalContainerEntityManagerFactoryBean` directly to enable classpath scanning for entity classes. See the [JavaDoc](#) for details.

5.10.3. CDI Integration

Instances of the repository interfaces are usually created by a container, for which Spring is the most natural choice when working with Spring Data. Spring offers sophisticated support for creating bean instances, as documented in [Creating Repository Instances](#). As of version 1.1.0, Spring Data JPA ships with a custom CDI extension that allows using the repository abstraction in CDI environments. The extension is part of the JAR. To activate it, include the Spring Data JPA JAR on your classpath.

You can now set up the infrastructure by implementing a CDI Producer for the `EntityManagerFactory` and `EntityManager`, as shown in the following example:

```
class EntityManagerFactoryProducer {

    @Produces
    @ApplicationScoped
    public EntityManagerFactory createEntityManagerFactory() {
        return Persistence.createEntityManagerFactory("my-persistence-unit");
    }

    public void close(@Disposes EntityManagerFactory entityManagerFactory) {
        entityManagerFactory.close();
    }

    @Produces
    @RequestScoped
    public EntityManager createEntityManager(EntityManagerFactory entityManagerFactory) {
        return entityManagerFactory.createEntityManager();
    }

    public void close(@Disposes EntityManager entityManager) {
        entityManager.close();
    }
}
```

The necessary setup can vary depending on the JavaEE environment. You may need to do nothing more than redeclare a `EntityManager` as a CDI bean, as follows:

```
class CdiConfig {

    @Produces
    @RequestScoped
    @PersistenceContext
    public EntityManager entityManager;
}
```

In the preceding example, the container has to be capable of creating JPA `EntityManager`s itself. All the configuration does is re-export the JPA `EntityManager` as a CDI bean.

The Spring Data JPA CDI extension picks up all available `EntityManager` instances as CDI beans and creates a proxy for a Spring Data repository whenever a bean of a repository type is requested by the container. Thus, obtaining an instance of a Spring Data repository is a matter of declaring an `@Injected` property, as shown in the following example:

```
class RepositoryClient {

    @Inject
    PersonRepository repository;

    public void businessMethod() {
        List<Person> people = repository.findAll();
    }
}
```

Appendix

Appendix A: Namespace reference

The `<repositories />` Element

The `<repositories />` element triggers the setup of the Spring Data repository infrastructure. The most important attribute is `base-package`, which defines the package to scan for Spring Data repository interfaces. See “[XML configuration](#)”. The following table describes the attributes of the `<repositories />` element:

Table 6. Attributes

Name	Description
<code>base-package</code>	Defines the package to be scanned for repository interfaces that extend <code>*Repository</code> (the actual interface is determined by the specific Spring Data module) in auto-detection mode. All packages below the configured package are scanned, too. Wildcards are allowed.
<code>repository-impl-postfix</code>	Defines the postfix to autodetect custom repository implementations. Classes whose names end with the configured postfix are considered as candidates. Defaults to <code>Impl</code> .
<code>query-lookup-strategy</code>	Determines the strategy to be used to create finder queries. See “ Query Lookup Strategies ” for details. Defaults to <code>create-if-not-found</code> .
<code>named-queries-location</code>	Defines the location to search for a Properties file containing externally defined queries.
<code>consider-nested-repositories</code>	Whether nested repository interface definitions should be considered. Defaults to <code>false</code> .

Appendix B: Populators namespace reference

The `<populator />` element

The `<populator />` element allows to populate the a data store via the Spring Data repository infrastructure.^[1]

Table 7. Attributes

Name	Description
<code>locations</code>	Where to find the files to read the objects from the repository shall be populated with.

Appendix C: Repository query keywords

Supported query keywords

The following table lists the keywords generally supported by the Spring Data repository query derivation mechanism. However, consult the store-specific documentation for the exact list of supported keywords, because some keywords listed here might not be supported in a particular store.

Table 8. Query keywords

Logical keyword	Keyword expressions
AND	And
OR	Or
AFTER	After, IsAfter
BEFORE	Before, IsBefore
CONTAINING	Containing, IsContaining, Contains
BETWEEN	Between, IsBetween
ENDING_WITH	EndingWith, IsEndingWith, EndsWith
EXISTS	Exists
FALSE	False, IsFalse
GREATER_THAN	GreaterThan, IsGreaterThan
GREATER_THAN_EQUALS	GreaterThanOrEqualTo, IsGreaterThanOrEqualTo
IN	In, IsIn
IS	Is, Equals, (or no keyword)
IS_EMPTY	IsEmpty, Empty
IS_NOT_EMPTY	IsNotEmpty, NotEmpty
IS_NOT_NULL	NotNull, IsNotNull
IS_NULL	Null, IsNull
LESS_THAN	LessThan, IsLessThan
LESS_THAN_EQUAL	LessThanOrEqualTo, IsLessThanOrEqualTo
LIKE	Like, IsLike

Logical keyword	Keyword expressions
NEAR	Near , IsNear
NOT	Not , IsNot
NOT_IN	NotIn , IsNotIn
NOT_LIKE	NotLike , IsNotLike
REGEX	Regex , MatchesRegex , Matches
STARTING_WITH	StartingWith , IsStartingWith , StartsWith
TRUE	True , IsTrue
WITHIN	Within , IsWithin

Appendix D: Repository query return types

Supported Query Return Types

The following table lists the return types generally supported by Spring Data repositories. However, consult the store-specific documentation for the exact list of supported return types, because some types listed here might not be supported in a particular store.



Geospatial types (such as `GeoResult`, `GeoResults`, and `GeoPage`) are available only for data stores that support geospatial queries.

Table 9. Query return types

Return type	Description
<code>void</code>	Denotes no return value.
Primitives	Java primitives.
Wrapper types	Java wrapper types.
<code>T</code>	An unique entity. Expects the query method to return one result at most. If no result is found, <code>null</code> is returned. More than one result triggers an <code>IncorrectResultSizeDataAccessException</code> .
<code>Iterator<T></code>	An <code>Iterator</code> .
<code>Collection<T></code>	A <code>Collection</code> .

Return type	Description
List<T>	A List.
Optional<T>	A Java 8 or Guava <code>Optional</code> . Expects the query method to return one result at most. If no result is found, <code>Optional.empty()</code> or <code>Optional.absent()</code> is returned. More than one result triggers an <code>IncorrectResultSizeDataAccessException</code> .
Option<T>	Either a Scala or Javaslang <code>Option</code> type. Semantically the same behavior as Java 8's <code>Optional</code> , described earlier.
Stream<T>	A Java 8 Stream.
Future<T>	A <code>Future</code> . Expects a method to be annotated with <code>@Async</code> and requires Spring's asynchronous method execution capability to be enabled.
CompletableFuture<T>	A Java 8 <code>CompletableFuture</code> . Expects a method to be annotated with <code>@Async</code> and requires Spring's asynchronous method execution capability to be enabled.
ListenableFuture	A <code>org.springframework.util.concurrent.ListenableFuture</code> . Expects a method to be annotated with <code>@Async</code> and requires Spring's asynchronous method execution capability to be enabled.
Slice	A sized chunk of data with an indication of whether there is more data available. Requires a <code>Pageable</code> method parameter.
Page<T>	A <code>Slice</code> with additional information, such as the total number of results. Requires a <code>Pageable</code> method parameter.
GeoResult<T>	A result entry with additional information, such as the distance to a reference location.
GeoResults<T>	A list of <code>GeoResult<T></code> with additional information, such as the average distance to a reference location.
GeoPage<T>	A <code>Page</code> with <code>GeoResult<T></code> , such as the average distance to a reference location.
Mono<T>	A Project Reactor <code>Mono</code> emitting zero or one element using reactive repositories. Expects the query method to return one result at most. If no result is found, <code>Mono.empty()</code> is returned. More than one result triggers an <code>IncorrectResultSizeDataAccessException</code> .
Flux<T>	A Project Reactor <code>Flux</code> emitting zero, one, or many elements using reactive repositories. Queries returning <code>Flux</code> can emit also an infinite number of elements.

Return type	Description
Single<T>	A RxJava <code>Single</code> emitting a single element using reactive repositories. Expects the query method to return one result at most. If no result is found, <code>Mono.empty()</code> is returned. More than one result triggers an <code>IncorrectResultSizeDataAccessException</code> .
Maybe<T>	A RxJava <code>Maybe</code> emitting zero or one element using reactive repositories. Expects the query method to return one result at most. If no result is found, <code>Mono.empty()</code> is returned. More than one result triggers an <code>IncorrectResultSizeDataAccessException</code> .
Flowable<T>	A RxJava <code>Flowable</code> emitting zero, one, or many elements using reactive repositories. Queries returning <code>Flowable</code> can emit also an infinite number of elements.

Appendix E: Frequently Asked Questions

Common

1. *I'd like to get more detailed logging information on what methods are called inside `JpaRepository` (for example). How can I gain them?*

You can make use of `CustomizableTraceInterceptor` provided by Spring, as shown in the following example:

```
<bean id="customizableTraceInterceptor" class="
    org.springframework.aop.interceptor.CustomizableTraceInterceptor">
    <property name="enterMessage" value="Entering ${methodName}(${arguments})"/>
    <property name="exitMessage" value="Leaving ${methodName}(): ${returnValue}"/>
</bean>

<aop:config>
    <aop:advisor advice-ref="customizableTraceInterceptor"
        pointcut="execution(public * org.springframework.data.jpa.repository.JpaRepository+.*(..))"/>
</aop:config>
```

Infrastructure

1. *Currently I have implemented a repository layer based on `HibernateDaoSupport`. I create a `SessionFactory` by using Spring's `AnnotationSessionFactoryBean`. How do I get Spring Data repositories working in this environment?*

You have to replace `AnnotationSessionFactoryBean` with the `HibernateJpaSessionFactoryBean`, as follows:

Example 111. Looking up a `SessionFactory` from a `HibernateEntityManagerFactory`

```
<bean id="sessionFactory"
    class="org.springframework.orm.jpa.vendor.HibernateJpaSessionFactoryBean">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>
```


Auditing

1. *I want to use Spring Data JPA auditing capabilities but have my database already configured to set modification and creation date on entities. How can I prevent Spring Data from setting the date programmatically.*

Set the `set-dates` attribute of the `auditing` namespace element to `false`.

Appendix F: Glossary

AOP

Aspect oriented programming

Commons DBCP

Commons DataBase Connection Pools - a library from the Apache foundation that offers pooling implementations of the DataSource interface.

CRUD

Create, Read, Update, Delete - Basic persistence operations.

DAO

Data Access Object - Pattern to separate persisting logic from the object to be persisted

Dependency Injection

Pattern to hand a component's dependency to the component from outside, freeing the component to lookup the dependent itself. For more information, see

http://en.wikipedia.org/wiki/Dependency_Injection.

EclipseLink

Object relational mapper implementing JPA - <http://www.eclipselink.org>

Hibernate

Object relational mapper implementing JPA - <http://www.hibernate.org>

JPA

Java Persistence API

Spring

Java application framework - <http://projects.spring.io/spring-framework>

1. see [XML configuration](#)