Spring Data JPA - Reference Documentation

Oliver Gierke, Thomas Darimont, Christoph Strobl, Mark Paluch

Version 2.1.0.M1, 2018-02-06

Table of Contents

Preface

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

Chapter 1. New & Noteworthy

1.1. What's new in Spring Data JPA 1.11

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

1.2. What's new in Spring Data JPA 1.10

- Support for Projections in repository query methods.
- Support for Query by Example.
- The following annotations have been enabled to build own, composed annotations: `@EntityGraph`, `@Lock`, `@Modifying`, `@Query`, `@QueryHints` and `@Procedure`.
- Support for `Contains` keyword on collection expressions.
- AttributeConverters for `ZoneId` of JSR-310 and ThreeTenBP.
- Upgrade to Querydsl 4, Hibernate 5, OpenJPA 2.4 and EclipseLink 2.6.1.

Chapter 2. Dependencies

Due to 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 by relying on the Spring Data Release Train BOM we ship with the compatible versions defined. In a Maven project you'd declare this dependency in the `<dependencyManagement />` section of your POM:

Example 1. Using the Spring Data release train BOM

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.data</groupId>
      <artifactId>spring-data-releasetrain</artifactId>
      <version>${release-train}</version>
      <scope>import</scope>
      <type>pom</type>
    </dependency>
  </dependencies>
</dependencyManagement>
```

The current release train version is `Lovelace-M1`. The train names are ascending alphabetically and currently available ones 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` etc. - milestones
- `RC1`, `RC2` etc. - release candidates
- `RELEASE` - GA release
- `SR1`, `SR2` etc. - service releases

A working example of using the BOMs can be found in our Spring Data examples repository. If that's in place declare the Spring Data modules you'd like to use without a version in the `<dependencies />` block.

Example 2. Declaring a dependency to a Spring Data module

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

2.1. Dependency management with Spring Boot

Spring Boot already selects a very recent version of Spring Data modules for you. In case you want to upgrade to a newer version nonetheless, simply configure the property `spring-data-releasetrain.version` to the train name and iteration you'd like to use.

2.2. Spring Framework

The current version of Spring Data modules require Spring Framework in version 5.0.3.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.

Chapter 3. Working with Spring Data Repositories

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

IMPORTANT

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. Adapt the XML namespace declaration and the types to be extended to the equivalents of the particular module that you are using. 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, consult the chapter on that module of this document.

3.1. Core concepts

The central interface in Spring Data repository abstraction is `Repository` (probably not that much of a surprise). 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); ②

  Iterable<T> findAll();               ③

  long count();                        ④

  void delete(T entity);               ⑤

  boolean existsById(ID primaryKey);   ⑥

  // … more functionality omitted.
}
```

① Saves the given entity.

② Returns the entity identified by the given id.

③ Returns all entities.

④ Returns the number of entities.

⑤ Deletes the given entity.

⑥ Indicates whether an entity with the given id exists.

NOTE We also provide persistence technology-specific abstractions like e.g. `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 like e.g. CrudRepository.

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

Example 4. PagingAndSortingRepository

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

  Iterable<T> findAll(Sort sort);

  Page<T> findAll(Pageable pageable);
}
```

Accessing the second page of `User` by a page size of 20 you could simply do something like this:

```
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.

Example 5. Derived Count Query

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

  long countByLastname(String lastname);
}
```

Example 6. Derived Delete Query

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

  long deleteByLastname(String lastname);

  List<User> removeByLastname(String lastname);
}
```

3.2. 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 will handle.

   ```
   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 via JavaConfig:

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

@EnableJpaRepositories
class Config {}
```

or via XML configuration:

```
<?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 are using the repository abstraction for any other store, you need to change this to the appropriate namespace declaration of your store module which should be exchanging `jpa` in favor of, for example, `mongodb`.

Also, note that the JavaConfig variant doesn't configure a package explictly as the package of the annotated class is used by default. To customize the package to scan use one of the `basePackage⋯` attribute of the data-store specific repository `@Enable⋯`-annotation.

4. Get the repository instance injected and use it.

```
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.

3.3. Defining repository interfaces

As a first step you 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`.

3.3.1. Fine-tuning repository definition

Typically, your repository interface will extend `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, simply copy the ones you want to expose from `CrudRepository` into your domain repository.

NOTE	This allows you to define your own abstractions on top of the provided Spring Data Repositories functionality.

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 this first step you defined a common base interface for all your domain repositories and exposed `findById(…)` as well as `save(…)`.These methods will be routed into the base repository implementation of the store of your choice provided by Spring Data ,e.g. in the case if JPA `SimpleJpaRepository`, because they are matching the method signatures in `CrudRepository`. So the `UserRepository` will now be able to save users, and find single ones by id, as well as triggering a query to find `Users` by their email address.

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

3.3.2. 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 to return other 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 will then be 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 using Spring Framework's nullability annotations. They provide a tooling-friendly approach and opt-in `null` checks during runtime:

- `@NonNullApi` – to be 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` – to be used on a parameter or return value that must not be `null` (not needed on parameter and return value where `@NonNullApi` applies).
- `@Nullable` – to be 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 allow tooling vendors like IDEA, Eclipse, or Kotlin to 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 package level using Spring's `@NonNullApi` in `package-info.java`:

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 will get validated at runtime for nullability constraints. Exceptions will be thrown in case a query execution result violates the defined constraint, i.e. the method would return `null` for some reason 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` that a method. Using the aforementioned result wrapper types will continue to work as expected, i.e. an empty result will be translated into the value representing absence.

Example 9. Using different nullability constraints

```
package com.acme;                                                 ①

import org.springframework.lang.Nullable;

interface UserRepository extends Repository<User, Long> {

  User getByEmailAddress(EmailAddress emailAddress);              ②

  @Nullable
  User findByEmailAddress(@Nullable EmailAddress emailAdress);    ③

  Optional<User> findOptionalByEmailAddress(EmailAddress emailAddress); ④
}
```

① The repository resides in a package (or sub-package) for which we've defined non-null behavior (see above).

② Will throw an `EmptyResultDataAccessException` in case the query executed does not produce a result. Will throw an `IllegalArgumentException` in case the `emailAddress` handed to the method is `null`.

③ Will return `null` in case the query executed does not produce a result. Also accepts `null` as value for `emailAddress`.

④ Will return `Optional.empty()` in case the query executed does not produce a result. Will throw an `IllegalArgumentException` in case 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 using method signatures but rather 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:

Example 10. Using nullability constraints on Kotlin repositories

```
interface UserRepository : Repository<User, String> {

  fun findByUsername(username: String): User     ①

  fun findByFirstname(firstname: String?): User? ②
}
```

① The method defines both, the parameter as non-nullable (the Kotlin default) as well as the result. The Kotlin compiler will already reject method invocations trying to hand `null` into the method. In case the query execution yields an empty result, an `EmptyResultDataAccessException` will be thrown.

② This method accepts `null` as parameter for `firstname` and returns `null` in case the query execution does not produce a result.

3.3.3. Using Repositories with multiple Spring Data modules

Using a unique Spring Data module in your application makes things simple hence, 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 case, it's required for a repository definition to distinguish between persistence technologies. Spring Data enters strict repository configuration mode because it detects multiple repository factories on the class path. Strict configuration requires 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's a valid candidate for the particular Spring Data module.
2. If the domain class is annotated with the module-specific type annotation, then it's a valid candidate for the particular Spring Data module. Spring Data modules accept either 3rd party annotations (such as JPA's `@Entity`) or provide own annotations such as `@Document` for Spring Data MongoDB/Spring Data Elasticsearch.

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.

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 using a unique Spring Data module, multiple modules cannot distinguish to which particular Spring Data these repositories should be bound.

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 annotation `@Entity` so this repository clearly belongs to Spring Data JPA. `UserRepository` uses `User` annotated with Spring Data MongoDB's `@Document` annotation.

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 identifying domain class annotations are used for strict repository

configuration identify repository candidates for a particular Spring Data module. Using multiple persistence technology-specific annotations on the same domain type is possible to reuse domain types across multiple persistence technologies, but then Spring Data is no longer able to determine a unique module to bind the repository.

The last way to distinguish repositories is scoping repository base packages. Base packages define the starting points for scanning for repository interface definitions which implies to have 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.

Example 15. Annotation-driven configuration of base packages

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

3.4. Defining query methods

The repository proxy has two ways to derive a store-specific query from the method name. It can derive the query from the method name directly, or by using a manually defined query. Available options depend on the actual store. However, there's got to be a strategy that decides what actual query is created. Let's have a look at the available options.

3.4.1. Query lookup strategies

The following strategies are available for the repository infrastructure to resolve the query. You can configure the strategy at the namespace through the `query-lookup-strategy` attribute in case of XML configuration or via the `queryLookupStrategy` attribute of the Enable${store}Repositories annotation in case of Java config. 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. Read more about query construction in Query creation.
- `USE_DECLARED_QUERY` tries to find a declared query and will throw an exception in case it can't 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 will be 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.

3.4.2. 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`.

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`, `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 support 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.

3.4.3. 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. Assume a `Person` has an `Address` with a `ZipCode`. In that case a method name of

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

creates the property traversal `x.address.zipCode`. The resolution algorithm starts with 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 continue building the tree down from there, splitting the tail up in the way just described. If the first split does not match, the algorithm move 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 and essentially choose the wrong property and finally 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 end up like so:

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

As we treat underscore as a reserved character we strongly advise to follow standard Java naming conventions (i.e. **not** using underscores in property names but camel case instead).

3.4.4. Special parameter handling

To handle parameters in your query you simply define method parameters as already seen in the examples above. Besides that the infrastructure will recognize certain specific types like `Pageable` and `Sort` to apply pagination and sorting to your queries dynamically.

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 allows you to 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, `Slice` can be used as return instead. A `Slice` only knows about whether there's a next `Slice` available which might be just sufficient when walking through a larger result set.

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

NOTE	To find out how many pages you get for a query entirely you have to trigger an additional count query. By default this query will be derived from the query you actually trigger.

3.4.5. Limiting query results

The results of query methods can be limited via the keywords `first` or `top`, which can be used interchangeably. An optional numeric value can be appended to top/first to specify the maximum result size to be returned. If the number is left out, a result size of 1 is assumed.

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 an `Optional` is supported.

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

NOTE Note that limiting the results in combination with dynamic sorting via a `Sort` parameter allows to express query methods for the 'K' smallest as well as for the 'K' biggest elements.

3.4.6. Streaming query results

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

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);
```

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

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

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

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

3.4.7. Async query results

Repository queries can be executed asynchronously using Spring's asynchronous method execution capability. This means the method will return immediately upon invocation and the actual query execution will occur in a task that has been submitted to a Spring TaskExecutor.

```
@Async
Future<User> findByFirstname(String firstname);               ①

@Async
CompletableFuture<User> findOneByFirstname(String firstname); ②

@Async
ListenableFuture<User> findOneByLastname(String lastname);    ③
```

① Use `java.util.concurrent.Future` as return type.

② Use a Java 8 `java.util.concurrent.CompletableFuture` as return type.

③ Use a `org.springframework.util.concurrent.ListenableFuture` as return type.

3.5. Creating repository instances

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

3.5.1. XML configuration

Each Spring Data module includes a repositories element that allows you to simply define a base package that Spring scans for you.

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 bean instances get created for. To do this you use `<include-filter />` and `<exclude-filter />` elements inside `<repositories />`. The semantics are exactly equivalent to the elements in Spring's context namespace. For details, see Spring reference documentation on these elements.

For example, to exclude certain interfaces from instantiation as repository, 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>
```

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

3.5.2. JavaConfig

The repository infrastructure can also be triggered using a store-specific

`@Enable${store}Repositories` annotation on a JavaConfig class. For an introduction into Java-based configuration of the Spring container, see the reference documentation. [1: JavaConfig in the Spring reference documentation]

A sample configuration to enable Spring Data repositories looks something like this.

Example 23. Sample annotation based repository configuration

```
@Configuration
@EnableJpaRepositories("com.acme.repositories")
class ApplicationConfiguration {

  @Bean
  EntityManagerFactory entityManagerFactory() {
    // …
  }
}
```

NOTE The sample 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. Consult the sections covering the store-specific configuration.

3.5.3. Standalone usage

You can also use the repository infrastructure outside of a Spring container, e.g. 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);
```

3.6. Custom implementations for Spring Data repositories

In this section you will learn about repository customization and how fragments form a composite repository.

When query method require a different behavior or can't be implemented by query derivation than it's necessary to provide a custom implementation. Spring Data repositories easily allow you to provide custom repository code and integrate it with generic CRUD abstraction and query

method functionality.

3.6.1. Customizing individual repositories

To enrich a repository with custom functionality, you first define a fragment interface and an implementation for the custom functionality. Then let your repository interface additionally extend from the fragment interface.

Example 25. Interface for custom repository functionality

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

Example 26. Implementation of custom repository functionality

```
class CustomizedUserRepositoryImpl implements CustomizedUserRepository {

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

NOTE The most important bit for the class to be found is the `Impl` postfix of the name on it compared to the fragment interface.

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

Example 27. Changes to your repository interface

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

  // Declare query methods here
}
```

Let your repository interface extend the fragment one. Doing so 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.

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 EmployeeRepository {

  void someEmployeeMethod(User user);

  User anotherEmployeeMethod(User user);
}

class ContactRepositoryImpl implements ContactRepository {

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

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

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 allows you to override base repository and aspect methods

and resolves ambiguity if two fragments contribute the same method signature. Repository fragments are not limited to be used in a single repository interface. Multiple repositories may use a fragment interface to reuse customizations across different repositories.

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
  }
}
```

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 we found a repository in. These classes need to follow the naming convention of appending the namespace element's attribute `repository-impl-postfix` to the found fragment interface name. This postfix defaults to `Impl`.

Example 32. Configuration example

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

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

The first configuration example will try to look up a class `com.acme.repository.CustomizedUserRepositoryImpl` to act as custom repository implementation,

whereas the second example will try to lookup `com.acme.repository.CustomizedUserRepositoryFooBar`.

Resolution of ambiguity

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

Given the following two custom implementations for the `CustomizedUserRepository` introduced above the first implementation will get picked. Its bean name is `customizedUserRepositoryImpl` matches that of the fragment interface (`CustomizedUserRepository`) plus the postfix `Impl`.

Example 33. Resolution of amibiguous 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` matches the one defined for the repository implementation in `com.acme.impl.two` and it will be picked instead of the first one.

Manual wiring

The approach just shown works well if your custom implementation uses annotation-based configuration and autowiring only, as it will be treated as any other Spring bean. If your implementation fragment bean needs special wiring, you simply declare the bean and name it after the conventions just described. The infrastructure will then refer to the manually defined bean definition by name instead of creating one itself.

Example 34. Manual wiring of custom implementations

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

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

3.6.2. Customize the base repository

The preceding approach requires customization of all repository interfaces when you want to customize the base repository behavior, so all repositories are affected. To change behavior for all repositories, you need to create an implementation that extends the persistence technology-specific repository base class. This class will then act as a custom base class for the repository proxies.

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
  }
}
```

WARNING The class needs to have a constructor of the super class which the store-specific repository factory implementation is using. In case the repository base class has multiple constructors, override the one taking an `EntityInformation` plus a store specific infrastructure object (e.g. an `EntityManager` or a template class).

The final step is to make the Spring Data infrastructure aware of the customized repository base class. In JavaConfig this is achieved by using the `repositoryBaseClass` attribute of the `@Enable…Repositories` annotation:

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.

Example 37. Configuring a custom repository base class using XML

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

3.7. 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 `@DomainEvents` you can use on a method of your aggregate root to make that publication as easy as possible.

Example 38. Exposing domain events from an aggregate root

```
class AnAggregateRoot {

    @DomainEvents ①
    Collection<Object> domainEvents() {
        // … return events you want to get published here
    }

    @AfterDomainEventPublication ②
    void callbackMethod() {
       // … potentially clean up domain events list
    }
}
```

① The method using `@DomainEvents` can either return a single event instance or a collection of events. It must not take any arguments.

② After all events have been published, a method annotated with `@AfterDomainEventPublication`. It e.g. can be used to potentially clean the list of events to be published.

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

3.8. 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.

3.8.1. Querydsl Extension

Querydsl is a framework which enables the construction of statically typed SQL-like queries via its fluent API.

Several Spring Data modules offer integration with Querydsl via `QueryDslPredicateExecutor`.

Example 39. QueryDslPredicateExecutor interface

```
public interface QueryDslPredicateExecutor<T> {

  Optional<T> findById(Predicate predicate);  ①

  Iterable<T> findAll(Predicate predicate);   ②

  long count(Predicate predicate);            ③

  boolean exists(Predicate predicate);        ④

  // … more functionality omitted.
}
```

① Finds and returns a single entity matching the `Predicate`.

② Finds and returns all entities matching the `Predicate`.

③ Returns the number of entities matching the `Predicate`.

④ Returns if an entity that matches the `Predicate` exists.

To make use of Querydsl support simply extend `QueryDslPredicateExecutor` on your repository interface.

Example 40. Querydsl integration on repositories

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

}
```

The above enables to write typesafe queries using Querydsl `Predicate` s.

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

userRepository.findAll(predicate);
```

3.8.2. Web support

NOTE	This section contains the documentation for the Spring Data web support as it is implemented as of Spring Data Commons in the 1.6 range. As it the newly introduced support changes quite a lot of things we kept the documentation of the former behavior in Legacy web support.

Spring Data modules ships with a variety of web support if the module supports the repository programming model. The web related stuff requires Spring MVC JARs on the classpath, some of them even provide integration with Spring HATEOAS [2: Spring HATEOAS - https://github.com/SpringSource/spring-hateoas]. In general, the integration support is enabled by using the `@EnableSpringDataWebSupport` annotation in your JavaConfig configuration class.

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 are using XML configuration, register either `SpringDataWebSupport` or `HateoasAwareSpringDataWebSupport` as Spring beans:

Example 42. Enabling Spring Data web support in XML

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

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

Basic web support

The configuration setup shown above will register a few basic components:

- A `DomainClassConverter` to enable Spring MVC to 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` allows you to use domain types in your Spring MVC controller method signatures directly, so that you don't have to manually lookup the instances via the repository:

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.

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

HandlerMethodArgumentResolvers for Pageable and Sort

The configuration snippet above also registers a `PageableHandlerMethodArgumentResolver` as well as an instance of `SortHandlerMethodArgumentResolver`. The registration enables `Pageable` and `Sort` being valid controller method arguments

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";
  }
}
```

This method signature will cause Spring MVC try to derive a Pageable instance from the request parameters 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 `property,property(,ASC|DESC)`. Default sort direction is ascending. Use multiple `sort` parameters if you want to switch directions, e.g. `?sort=firstname&sort=lastname,asc`.

To customize this behavior register a bean implementing the interface `PageableHandlerMethodArgumentResolverCustomizer` or `SortHandlerMethodArgumentResolverCustomizer` respectively. It's `customize()` method will get called allowing you to change settings. Like in the following example.

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

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

In case 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}_`. So for a method signature like this:

```
String showUsers(Model model,
      @Qualifier("foo") Pageable first,
      @Qualifier("bar") Pageable second) { … }
```

you have to populate `foo_page` and `bar_page` etc.

The default `Pageable` handed into the method is equivalent to a `new PageRequest(0, 20)` but can be customized 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, the `PagedResourcesAssembler`.

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 above allows the `PagedResourcesAssembler` to be used as controller method argument. Calling `toResources(…)` on it will cause the following:

- The content of the `Page` will become the content of the `PagedResources` instance.
- The `PagedResources` will get a `PageMetadata` instance attached populated with information form the `Page` and the underlying `PageRequest`.
- The `PagedResources` gets `prev` and `next` links attached depending on the page's state. The links will point to the URI the method invoked is mapped to. The pagination parameters added to the method will match the setup of the `PageableHandlerMethodArgumentResolver` to make sure the links can be resolved later on.

Assume we have 30 Person instances in the database. You can now trigger a request `GET http://localhost:8080/persons` and you'll see something similar to this:

```
{ "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 picks up the default configuration present to resolve the parameters into a `Pageable` for an upcoming request. This means, if you change that configuration, the links will 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 to overloads of the `PagedResourcesAssembler.toResource(···)` method.

Web databinding support

Spring Data projections – generally described in Projections – can be used to bind incoming request payloads by either using JSONPath expressions (requires Jayway JasonPath or XPath expressions (requires XmlBeam).

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 above can be used as Spring MVC handler method argument or via `ParameterizedTypeReference` on one of `RestTemplate`'s methods. The method declarations above would try to find `firstname` anywhere in the given document. The `lastname` XML looup 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 in case the former doesn't return a value. That way changes if the structure of the source document can be mitigated easily without having to touch 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.

This means that given the `User` object from previous samples a query string

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

can be resolved to

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

using the `QuerydslPredicateArgumentResolver`.

NOTE	The feature will be automatically enabled along `@EnableSpringDataWebSupport` when Querydsl is found on the classpath.

Adding a `@QuerydslPredicate` to the method signature will provide a ready to use `Predicate` which can be executed via the `QueryDslPredicateExecutor`.

TIP	Type information is typically resolved from the methods return type. Since those information does not necessarily match the domain type it might be a good idea to use the `root` attribute of `QuerydslPredicate`.

```
@Controller
class UserController {

  @Autowired UserRepository repository;

  @RequestMapping(value = "/", method = RequestMethod.GET)
  String index(Model model, @QuerydslPredicate(root = User.class) Predicate
predicate,    ①
          Pageable pageable, @RequestParam MultiValueMap<String, String>
parameters) {

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

    return "index";
  }
}
```

① 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 via the `bindings` attribute of `@QuerydslPredicate` or by making use of Java 8 `default methods` adding the `QuerydslBinderCustomizer` to the repository interface.

```
interface UserRepository extends CrudRepository<User, String>,
                                 QueryDslPredicateExecutor<User>,
①
                                 QuerydslBinderCustomizer<QUser> {
②

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

    bindings.bind(user.username).first((path, value) -> path.contains(value))
③
    bindings.bind(String.class)
      .first((StringPath path, String value) -> path.containsIgnoreCase(value));
④
    bindings.excluding(user.password);
⑤
  }
}
```

① `QueryDslPredicateExecutor` provides access to specific finder methods for `Predicate`.

② `QuerydslBinderCustomizer` defined on the repository interface will be 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.

3.8.3. Repository populators

If you work with the Spring JDBC module, you probably are familiar with the support to populate a `DataSource` using 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 easily 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 , do 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>
```

This declaration causes the `data.json` file to be read and deserialized via a Jackson `ObjectMapper`.

The type to which the JSON object will be unmarshalled to will be determined by inspecting the `\_class` attribute of the JSON document. The infrastructure will eventually select the appropriate repository to handle the object just deserialized.

To rather use XML to define the data the repositories shall be populated with, you can use the `unmarshaller-populator` element. You configure it to use one of the XML marshaller options Spring OXM provides you with. See the Spring reference documentation for details.

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>
```

3.8.4. Legacy web support

Domain class web binding for Spring MVC

Given you are developing a Spring MVC web application you typically have to resolve domain class ids from URLs. By default your task is to transform that request parameter or URL part into the domain class to hand it to layers below then or execute business logic on the entities directly. This would look something like this:

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

  private final UserRepository userRepository;

  UserController(UserRepository userRepository) {
    Assert.notNull(repository, "Repository must not be null!");
    this.userRepository = userRepository;
  }

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

    // Do null check for id
    User user = userRepository.findById(id);
    // Do null check for user

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

First you declare a repository dependency for each controller to look up the entity managed by the controller or repository respectively. Looking up the entity is boilerplate as well, as it's always a `findById(…)` call. Fortunately Spring provides means to register custom components that allow conversion between a `String` value to an arbitrary type.

PropertyEditors

For Spring versions before 3.0 simple Java `PropertyEditors` had to be used. To integrate with that, Spring Data offers a `DomainClassPropertyEditorRegistrar`, which looks up all Spring Data repositories registered in the `ApplicationContext` and registers a custom `PropertyEditor` for the managed domain class.

```
<bean class="….web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">
  <property name="webBindingInitializer">
    <bean class="….web.bind.support.ConfigurableWebBindingInitializer">
      <property name="propertyEditorRegistrars">
        <bean class=
"org.springframework.data.repository.support.DomainClassPropertyEditorRegistrar" />
      </property>
    </bean>
  </property>
</bean>
```

If you have configured Spring MVC as in the preceding example, you can configure your controller as follows, which reduces a lot of the clutter and boilerplate.

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

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

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

Reference Documentation

Chapter 4. JPA Repositories

This chapter will point out the specialties for repository support for JPA. This builds on the core repository support explained in Working with Spring Data Repositories. So make sure you've got a sound understanding of the basic concepts explained there.

4.1. Introduction

4.1.1. 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 using the `repositories` element:

Example 50. Setting up JPA repositories 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 this 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 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 `EntityManagerFactory` to be used with the repositories being detected by the `repositories` element. Usually used if multiple `EntityManagerFactory` beans are used within the application. If not configured we will automatically lookup the `EntityManagerFactory` bean with the name `entityManagerFactory` in the `ApplicationContext`.

`transaction-manager-ref`	Explicitly wire the `PlatformTransactionManager` to be used with the repositories being detected by the `repositories` element. Usually only necessary if multiple transaction managers and/or `EntityManagerFactory` beans have been configured. Default to a single defined `PlatformTransactionManager` inside the current `ApplicationContext`.

Note that we require a `PlatformTransactionManager` bean named `transactionManager` to be present if no explicit `transaction-manager-ref` is defined.

4.1.2. Annotation based configuration

The Spring Data JPA repositories support cannot only be activated through an XML namespace but also using an annotation through JavaConfig.

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() {

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

NOTE It's important to create `LocalContainerEntityManagerFactoryBean` and not `EntityManagerFactory` directly since the former also participates in exception translation mechanisms besides simply creating `EntityManagerFactory`.

The just shown configuration class sets up an embedded HSQL database using the `EmbeddedDatabaseBuilder` API of spring-jdbc. We then set up a `EntityManagerFactory` and use Hibernate as sample persistence provider. The last infrastructure component declared here is the `JpaTransactionManager`. We finally activate Spring Data JPA repositories using the `@EnableJpaRepositories` annotation which essentially carries the same attributes as the XML namespace does. If no base package is configured it will use the one the configuration class resides in.

4.2. Persisting entities

4.2.1. Saving entities

Saving an entity can be performed via the `CrudRepository.save(…)`-Method. It will persist or merge the given entity using the underlying JPA `EntityManager`. If the entity has not been persisted yet Spring Data JPA will save the entity via a call to the `entityManager.persist(…)` method, otherwise the `entityManager.merge(…)` method will be called.

Entity state detection strategies

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

Table 3. Options for detection whether an entity is new in Spring Data JPA

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 will be assumed as new, otherwise as not new.
Implementing `Persistable`	If an entity implements `Persistable`, Spring Data JPA will delegate 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.

4.3. Query methods

4.3.1. Query lookup strategies

The JPA module supports defining a query manually as String or have 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 NamedQueries for more information) or rather annotate your query method with `@Query` (see Using @Query for details).

4.3.2. Query creation

Generally the query creation mechanism for JPA works as described in Query methods. Here's a short example of 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 will 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 will do a property check and traverse nested properties as described in Property expressions. Here's an overview of the keywords supported for JPA and what a method containing that keyword essentially translates to.

Table 4. Supported keywords inside method names

Keyword	Sample	JPQL snippet
`And`	`findByLastnameAndFirstname`	`… where x.lastname = ?1 and x.firstname = ?2`
`Or`	`findByLastnameOrFirstname`	`… where x.lastname = ?1 or x.firstname = ?2`
`Is,Equals`	`findByFirstname,findByFirstnameIs,findByFirstnameEquals`	`… where x.firstname = ?1`
`Between`	`findByStartDateBetween`	`… where x.startDate between ?1 and ?2`
`LessThan`	`findByAgeLessThan`	`… where x.age < ?1`
`LessThanEqual`	`findByAgeLessThanEqual`	`… where x.age <= ?1`
`GreaterThan`	`findByAgeGreaterThan`	`… where x.age > ?1`
`GreaterThanEqual`	`findByAgeGreaterThanEqual`	`… where x.age >= ?1`
`After`	`findByStartDateAfter`	`… where x.startDate > ?1`
`Before`	`findByStartDateBefore`	`… where x.startDate < ?1`
`IsNull`	`findByAgeIsNull`	`… where x.age is null`
`IsNotNull,NotNull`	`findByAge(Is)NotNull`	`… where x.age not null`
`Like`	`findByFirstnameLike`	`… where x.firstname like ?1`
`NotLike`	`findByFirstnameNotLike`	`… where x.firstname not like ?1`
`StartingWith`	`findByFirstnameStartingWith`	`… where x.firstname like ?1` (parameter bound with appended `%`)
`EndingWith`	`findByFirstnameEndingWith`	`… where x.firstname like ?1` (parameter bound with prepended `%`)
`Containing`	`findByFirstnameContaining`	`… where x.firstname like ?1` (parameter bound wrapped in `%`)
`OrderBy`	`findByAgeOrderByLastnameDesc`	`… where x.age = ?1 order by x.lastname desc`
`Not`	`findByLastnameNot`	`… where x.lastname <> ?1`
`In`	`findByAgeIn(Collection<Age> ages)`	`… where x.age in ?1`

Keyword	Sample	JPQL snippet
NotIn	findByAgeNotIn(Collection<Age> ages)	… where x.age not in ?1
True	findByActiveTrue()	… where x.active = true
False	findByActiveFalse()	… where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	… where UPPER(x.firstame) = UPPER(?1)

NOTE In and NotIn also take any subclass of Collection as parameter as well as arrays or varargs. For other syntactical versions of the very same logical operator check Repository query keywords.

4.3.3. Using JPA NamedQueries

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

XML named query definition

To use XML configuration simply add the necessary <named-query /> element to the orm.xml JPA configuration file located in 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>
```

As you can see the query has a special name which will be used to resolve it at runtime.

Annotation configuration

Annotation configuration has the advantage of not needing another configuration file to be edited, probably lowering maintenance costs. 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 all you need to do is to 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 will try 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 example here would use the named queries defined above instead of trying to create a query from the method name.

4.3.4. 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 actually can bind them directly using the Spring Data JPA `@Query` annotation rather than annotating them to the domain class. This will free the domain class from persistence specific information and co-locate the query to the repository interface.

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

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 using @Query allows the definition of advanced `LIKE` expressions inside the query definition.

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 just shown sample `LIKE` delimiter character `%` is recognized and the query transformed into a valid JPQL query (removing the `%`). Upon query execution the parameter handed into the method call gets augmented with the previously recognized `LIKE` pattern.

Native queries

The `@Query` annotation allows to execute native queries by setting the `nativeQuery` flag to true.

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);
}
```

Note, that we currently don't support execution of dynamic sorting for native queries as we'd have to manipulate the actual query declared and we cannot do this reliably for native SQL. You can however use native queries for pagination by specifying the count query yourself:

Example 59. Declare native count queries for pagination at the query method 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);
}
```

This also works with named native queries by adding the suffix `.count` to a copy of your query. Be aware that you probably must register a result set mapping for your count query, though.

4.3.5. Using Sort

Sorting can be done be either providing a `PageRequest` or using `Sort` directly. The properties actually used within the `Order` instances of `Sort` need to match to 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*.

NOTE Using any non referenceable path expression leads to an Exception.

Using `Sort` together with `@Query` however allows you to sneak in non path checked `Order` instances containing *functions* within the `ORDER BY` clause. This is possible because the `Order` is just appended to the given query string. By default we will reject any `Order` instance containing function calls, but you can use `JpaSort.unsafe` to add potentially unsafe ordering.

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"));               ①
repo.findByAndSort("stark", new Sort("LENGTH(firstname)"));           ②
repo.findByAndSort("targaryen", JpaSort.unsafe("LENGTH(firstname)")); ③
repo.findByAsArrayAndSort("bolton", new Sort("fn_len"));              ④
```

① Valid `Sort` expression pointing to property in domain model.

② Invalid `Sort` containing function call. Thows Exception.

③ Valid `Sort` containing explicitly *unsafe* `Order`.

④ Valid `Sort` expression pointing to aliased function.

4.3.6. Using named parameters

By default Spring Data JPA will use position based parameter binding as described in all the samples above. This makes query methods a little error prone to 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.

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);
}
```

Note that the method parameters are switched according to the occurrence in the query defined.

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

4.3.7. Using SpEL expressions

As of Spring Data JPA release 1.4 we support the usage of restricted SpEL template expressions in manually defined queries via `@Query`. Upon query execution these expressions are evaluated against a predefined set of variables. We support the following list of variables to be used in a manual query.

Table 5. Supported variables inside SpEL based query templates

Variable	Usage	Description
`entityName`	`select x from #{#entityName} x`	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 then it will be used. Otherwise the simple class-name of the domain type will be 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 with a manually defined query. In order not to have to state the actual entity name in the query string of a `@Query` annotation one can use the `#{#entityName}` Variable.

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

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);
}
```

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` will pick up potential future remappings of the User class to a different entity name (e.g. 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. In order not to have to 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.

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

4.3.8. Modifying queries

All the sections above describe how to declare queries to access a given entity or collection of entities. Of course you can add custom modifying behaviour by using facilities described in Custom implementations for Spring Data repositories. As this approach is feasible for comprehensive custom functionality, you can achieve the execution of modifying queries that actually only need parameter binding by annotating the query method with `@Modifying`:

Example 64. Declaring manipulating queries

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

This will trigger the query annotated to the method as 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 JavaDoc of `EntityManager.clear()` for details) since this will effectively drop all non-flushed changes still pending in the `EntityManager`. If you wish the `EntityManager` to be cleared automatically you can set `@Modifying` annotation's `clearAutomatically` attribute to `true`.

Derived delete queries

Spring Data JPA also supports derived delete queries that allow you to avoid having to declare the JPQL query explicitly.

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's basically producing 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 will issue a single JPQL query (i.e. the one defined in the annotation) against the database. This means, even currently loaded instances of `User` won't see lifecycle callbacks invoked.

To make sure lifecycle queries are actually invoked, an invocation of `deleteByRoleId(…)` will actually execute a query and then deleting 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 keep behavior in sync with the implementations of other `delete(…)` methods in `CrudRepository`.

4.3.9. 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 addtional count query triggered when applying pagination.

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 just shown 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.

4.3.10. Configuring Fetch- and LoadGraphs

The JPA 2.1 specification introduced support for specifiying Fetch- and LoadGraphs that we also support via the `@EntityGraph` annotation which allows to reference a `@NamedEntityGraph` definition, that can be annotated on an entity, to be used to configure the fetch plan of the resulting query. The type (Fetch / Load) of the fetching can be configured via the `type` attribute on the `@EntityGraph` annotation. Please have a look at the JPA 2.1 Spec 3.7.4 for further reference.

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>();

  …
}
```

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

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

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

}
```

It is also possible to define *ad-hoc* entity graphs via `@EntityGraph`. The provided `attributePaths` will be translated into the according `EntityGraph` without the need of having to explicitly add `@NamedEntityGraph` to your domain types.

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 getByGroupName(String name);

}
```

4.3.11. 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 rather project on certain attributes of those types. Spring Data allows to model dedicated return types to more selectively retrieve partial views onto the managed aggregates.

Imagine a sample repository and aggregate root type like this:

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 we'd want to retrieve the person's name attributes only. What means does Spring Data offer to achieve this?

Interface-based projections

The easiest way to limit the result of the queries to expose the name attributes only is by declaring an interface that will expose accessor methods for the properties to be read:

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. This allows a query method to be added like this:

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 will create proxy instances of that interface at runtime for each element returned and forward calls to the exposed methods to the target object.

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

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 will be obtained and wrapped into a projecting proxy in turn.

Closed projections

A projection interface whose accessor methods all match properties of the target aggregate are considered closed projections.

Example 74. A closed projection

```
interface NamesOnly {

  String getFirstname();
  String getLastname();
}
```

If a closed projection is used, Spring Data modules can even optimize the query execution as we exactly know about all attributes that are needed to back the projection proxy. For more details on that, please refer to 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 on it:

Example 75. An Open Projection

```
interface NamesOnly {

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

The aggregate root backing the projection is available via the `target` variable. A projection interface using `@Value` an open projection. Spring Data won't be able to apply query execution optimizations in this case as the SpEL expression could use any attributes of the aggregate root.

The expressions used in `@Value` shouldn't become too complex as you'd want to avoid programming in `String`s. For very simple expressions, one option might be to resort to default methods:

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 simply invoke that from the SpEL expression:

Example 77. Sample Person object

```
@Component
class MyBean {

  String getFullName(Person person) {
    …
  }
}

interface NamesOnly {

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

Note, how the SpEL expression refers to `myBean` and invokes the `getFullName(…)` method forwarding the projection target as 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 via an `Object` array named `args`.

Example 78. Sample Person object

```
interface NamesOnly {

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

Again, for more complex expressions rather use a Spring bean and let the expression just invoke a method as described above.

Class-based projections (DTOs)

Another way of defining projections is using value type DTOs that hold properties for the fields that are supposed to be retrieved. These DTO types can be used exactly the same way projection interfaces are used, except that no proxying is going on here and no nested projections can be applied.

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

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
}
```

TIP

Avoiding boilerplate code for projection DTOs

The code that needs to be written for a DTO can be dramatically simplified using Project Lombok, which provides an `@Value` annotation (not to mix up with Spring's `@Value` annotation shown in the interface examples above). The sample DTO above would become this:

```
@Value
class NamesOnly {
    String firstname, lastname;
}
```

Fields are private final by default, the class exposes a constructor taking 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, it might be desirable to rather select the type to be used at invocation time. To apply dynamic projections, use a query method like this:

Example 80. A repository using a dynamic projection parameter

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

  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:

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);
}
```

4.4. Stored procedures

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

repository method.

Example 82. The definition of the pus1inout 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 via 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 {}
```

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

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

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

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

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

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

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

Example 87. Referencing implicitly mapped named stored procedure "User.plus1" in EntityManager via method-name.

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

4.5. Specifications

JPA 2 introduces a criteria API that can be used to build queries programmatically. Writing a `criteria` you actually define the where-clause of a query for a domain class. Taking another step back these criteria can be regarded as 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 using the JPA criteria API. To support specifications you can extend your repository interface with the `JpaSpecificationExecutor` interface:

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

The additional interface carries methods that allow you to execute specifications in a variety of ways. For example, the `findAll` method will return all entities that match the specification:

```
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);
}
```

Okay, so what is the typical use case? 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. Here's an 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 will hopefully be reduced by Java 8 closures) but the client side becomes much nicer as you will see below. The `_Customer` type is a metamodel type generated using the JPA Metamodel generator (see the Hibernate implementation's documentation for example). So the expression `_Customer.createdAt` is asuming the `Customer` having 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());
```

Okay, why not simply create a query for this kind of data access? You're right. 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 like this:

Example 90. Combined Specifications

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

As you can see, `Specifications` offers some glue-code methods to chain and combine `Specification` instances. Thus extending your data access layer is just a matter of creating new `Specification` implementations and combining them with ones already existing.

4.6. Query by Example

4.6.1. Introduction

This chapter will give you an introduction to Query by Example and explain how to use Examples.

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

4.6.2. Usage

The Query by Example API consists of three parts:

- Probe: That is 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 suited for several use-cases but also comes with limitations:

When to use

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

Limitations

- No support for nested/grouped property constraints like `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, simply create an interface for your repository:

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
}
```

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

Example 92. Simple Example

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

Example<Person> example = Example.of(person);          ③
```

① Create a new instance of the domain object

② Set the properties to query

③ Create the `Example`

Examples are ideally be executed with repositories. To do so, let your repository interface extend `QueryByExampleExecutor<T>`. Here's 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.
}
```

4.6.3. Example matchers

Examples are not limited to default settings. You can specify own defaults for string matching, null handling and property-specific settings using the `ExampleMatcher`.

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's usable at this stage even without further configuration.

④ Construct a new `ExampleMatcher` to ignore the property path `lastname`.

⑤ Construct a new `ExampleMatcher` to ignore the property path `lastname` and to include null values.

⑥ Construct a new `ExampleMatcher` to ignore the property path `lastname`, to include null values, and use perform suffix string matching.

⑦ Create a new `Example` based on the domain object and the configured `ExampleMatcher`.

By default the `ExampleMatcher` will expect 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 (e.g. "firstname" and "lastname", "address.city" for nested properties). You can tune it with matching options and case sensitivity.

Example 95. Configuring matcher options

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

Another style to configure matcher options is by using Java 8 lambdas. This approach is a callback that asks the implementor to modify the matcher. It's not required to return the matcher because configuration options are held within the matcher instance.

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 `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.

Table 6. Scope of `ExampleMatcher` settings

Setting	Scope
Null-handling	`ExampleMatcher`
String matching	`ExampleMatcher` and property path
Ignoring properties	Property path
Case sensitivity	`ExampleMatcher` and property path
Value transformation	Property path

4.6.4. Executing an example

In Spring Data JPA you can use Query by Example with Repositories.

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));
  }
}
```

NOTE Only `SingularAttribute` properties can currently be used for property matching.

Property specifier accepts property names (e.g. "firstname" and "lastname"). You can navigate by chaining properties together with dots ("address.city"). You can tune it with matching options and case sensitivity.

Table 7. `StringMatcher` options

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

4.7. Transactionality

CRUD methods on repository instances are transactional by default. For reading 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 simply 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
}
```

This will cause the `findAll()` method to be executed with a timeout of 10 seconds and without the `readOnly` flag.

Another possibility to alter transactional behaviour is using a facade or service implementation that typically covers more than one repository. Its purpose is to define transactional boundaries for non-CRUD operations:

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 will cause call to `addRoleToAllUsers(…)` to run inside a transaction (participating in an existing one or create a new one if none already running). The transaction configuration at the repositories will be neglected then as the outer transaction configuration determines the actual one used. Note that you will have to activate `<tx:annotation-driven />` or use `@EnableTransactionManagement` explicitly to get annotation based configuration at facades working. The example above assumes you are using component scanning.

4.7.1. Transactional query methods

To allow your query methods to be transactional simply use `@Transactional` at the repository interface you define.

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

NOTE

It's definitely reasonable to use transactions for read only queries and we can mark them as such by setting the `readOnly` flag. This will not, however, act as 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 instead is propagated as hint to the underlying JDBC driver for performance optimizations. Furthermore, Spring will perform some optimizations on the underlying JPA provider. E.g. 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).

4.8. Locking

To specify the lock mode to be used the `@Lock` annotation can be used on query methods:

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 will cause the query being triggered to be equipped with the `LockModeType READ`. You can also define locking for CRUD methods by redeclaring them in your repository interface and adding the `@Lock` annotation:

Example 102. Defining lock metadata on CRUD methods

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

  // Redeclaration of a CRUD method
  @Lock(LockModeType.READ);
  List<User> findAll();
}
```

4.9. Auditing

4.9.1. Basics

Spring Data provides sophisticated support to transparently keep track of who created or changed an entity and the point in time this 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`, `@LastModifiedBy` to capture the user who created or modified the entity as well as `@CreatedDate` and `@LastModifiedDate` to capture the point in time this 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'd like to capture. For the annotations capturing the points in time can be used on properties of type JodaTimes `DateTime`, legacy Java `Date` and `Calendar`, JDK8 date/time types as well as `long`/`Long`.

Interface-based auditing metadata

In case you don't 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's also a convenience base class `AbstractAuditable` which you can extend to avoid the need to manually implement the interface methods. Be aware that this 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 of what type the properties annotated with `@CreatedBy` or `@LastModifiedBy` have to be.

Here's an example implementation of the interface using 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 is accessing the `Authentication` object provided by Spring Security and looks up the custom `UserDetails` instance from it that you have created in your `UserDetailsService` implementation. We're assuming here that you are exposing the domain user through that `UserDetails` implementation but you could also look it up from anywhere based on the `Authentication` found.

4.10. JPA Auditing

4.10.1. General auditing configuration

Spring Data JPA ships with an entity listener that can be used to trigger capturing auditing information. So first you have to register the `AuditingEntityListener` inside your `orm.xml` to be used for all entities in your persistence contexts:

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` per entity using the `@EntityListeners` annotation:

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

}
```

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

With that in place, activating auditing functionality is just a matter of adding the Spring Data JPA `auditing` namespace element to your configuration:

Example 106. Activating auditing using XML configuration

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

As of Spring Data JPA 1.5, auditing can be enabled by annotating a configuration class with the @EnableJpaAuditing annotation.

Example 107. Activating auditing via 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 will pick it up automatically and use 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`.

Chapter 5. Miscellaneous

5.1. Using JpaContext in custom implementations

When working with multiple `EntityManager` instances and custom repository implementations you'll need to make sure you wire the correct `EntityManager` into the repository implementation class. This could be solved by explicitly naming the `EntityManager` in the `@PersistenceContext` annotation or using `@Qualifier` in case the `EntityManager` is injected via `@Autowired`.

As of Spring Data JPA 1.9, we ship a class `JpaContext` that allows to obtain the `EntityManager` by managed domain class assuming it's only managed by one of the `EntityManager` instances in the application.

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);
  }

  …
}
```

This approach has the advantage that the repository does not have to be touched to alter the reference to the persistence unit in case the domain type gets assigned to a different persistence unit.

5.2. Merging persistence units

Spring supports having multiple persistence units out of the box. Sometimes, however, you might want to modularize your application but still make sure that all these modules run inside a single persistence unit at runtime. To do so Spring Data JPA offers a `PersistenceUnitManager` implementation that automatically merges persistence units based on their name.

Example 109. Using MergingPersistenceUnitmanager

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

5.2.1. Classpath scanning for @Entity classes and JPA mapping files

A plain JPA setup requires all annotation mapped entity classes listed in `orm.xml`. 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 will then scan the given package for classes annotated with @Entity or @MappedSuperclass and also loads the configuration files matching the filename pattern and hands them to the JPA configuration. The PostProcessor has to be configured like this:

Example 110. Using ClasspathScanningPersistenceUnitPostProcessor

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

NOTE 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.

5.3. CDI integration

Instances of the repository interfaces are usually created by a container, which Spring is the most natural choice when working with Spring Data. There's sophisticated support to easily set up Spring to create bean instances 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 so all you need to do to activate it is dropping the Spring Data JPA JAR into your classpath.

You can now set up the infrastructure by implementing a CDI Producer for the `EntityManagerFactory` and `EntityManager`:

```
class EntityManagerFactoryProducer {

  @Produces
  @ApplicationScoped
  public EntityManagerFactory createEntityManagerFactory() {
    return Persistence.createEntityManagerFactory("my-presistence-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 run in. It might also just be enough to redeclare a `EntityManager` as CDI bean as follows:

```
class CdiConfig {

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

In this example, the container has to be capable of creating JPA `EntityManagers` itself. All the configuration does is re-exporting the JPA `EntityManager` as CDI bean.

The Spring Data JPA CDI extension will pick up all EntityManagers availables as CDI beans and create a proxy for a Spring Data repository whenever an 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:

```
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. [3: see XML configuration]

Table 8. Attributes

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

Appendix B: Populators namespace reference

The <populator /> element

The `<populator />` element allows to populate the a data store via the Spring Data repository infrastructure. [4: see XML configuration]

Table 9. Attributes

Name	Description
`locations`	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 listed here might not be supported in a particular store.

Table 10. 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`	`GreaterThanEqual`, `IsGreaterThanEqual`
`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`	`LessThanEqual`, `IsLessThanEqual`
`LIKE`	`Like`, `IsLike`
`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 listed here might not be supported in a particular store.

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

Table 11. Query return types

Return type	Description
`void`	Denotes no return value.
Primitives	Java primitives.
Wrapper types	Java wrapper types.
`T`	An unique entity. Expects the query method to return one result at most. In case no result is found `null` is returned. More than one result will trigger an `IncorrectResultSizeDataAccessException`.
`Iterator<T>`	An `Iterator`.
`Collection<T>`	A `Collection`.
`List<T>`	A `List`.
`Optional<T>`	A Java 8 or Guava `Optional`. Expects the query method to return one result at most. In case no result is found `Optional.empty()` /`Optional.absent()` is returned. More than one result will trigger an `IncorrectResultSizeDataAccessException`.
`Option<T>`	An either Scala or JavaSlang `Option` type. Semantically same behavior as Java 8's `Optional` described above.
`Stream<T>`	A Java 8 `Stream`.
`Future<T>`	A `Future`. Expects method to be annotated with `@Async` and requires Spring's asynchronous method execution capability enabled.
`CompletableFuture<T>`	A Java 8 `CompletableFuture`. Expects method to be annotated with `@Async` and requires Spring's asynchronous method execution capability enabled.
`ListenableFuture`	A `org.springframework.util.concurrent.ListenableFuture`. Expects method to be annotated with `@Async` and requires Spring's asynchronous method execution capability enabled.
`Slice`	A sized chunk of data with information whether there is more data available. Requires a `Pageable` method parameter.
`Page<T>`	A `Slice` with additional information, e.g. the total number of results. Requires a `Pageable` method parameter.
`GeoResult<T>`	A result entry with additional information, e.g. distance to a reference location.

Return type	Description
`GeoResults<T>`	A list of `GeoResult<T>` with additional information, e.g. average distance to a reference location.
`GeoPage<T>`	A `Page` with `GeoResult<T>`, e.g. average distance to a reference location.

Appendix E: Frequently asked questions

Common

I'd like to get more detailed logging information on what methods are called inside `JpaRepository`, e.g. How can I gain them?

You can make use of `CustomizableTraceInterceptor` provided by Spring:

```
<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

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

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

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

Appendix F: Glossary

AOP

Aspect oriented programming

Commons DBCP

Commons DataBase Connection Pools - Library of the Apache foundation offering 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 dependant 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