

Spring Data for Apache Solr

Christoph Strobl, Oliver Gierke, Mark Pollack, Thomas Risberg

Version 3.0.5.RELEASE, 2018-02-28

Table of Contents

Preface	2
Project Metadata	3
Requirements	4
1. New & Noteworthy	5
1.1. What's new in Spring Data for Apache Solr 2.1	5
1.2. What's new in Spring Data for Apache Solr 2.0	5
1.3. What's new in Spring Data for Apache Solr 1.5	5
1.4. What's new in Spring Data for Apache Solr 1.4	5
2. Working with Spring Data Repositories	6
2.1. Core concepts	6
2.2. Query methods	8
2.3. Defining repository interfaces	10
2.3.1. Fine-tuning repository definition	10
2.3.2. Null handling of repository methods	10
2.3.3. Using Repositories with multiple Spring Data modules	13
2.4. Defining query methods	16
2.4.1. Query lookup strategies	16
2.4.2. Query creation	17
2.4.3. Property expressions	18
2.4.4. Special parameter handling	18
2.4.5. Limiting query results	19
2.4.6. Streaming query results	20
2.4.7. Async query results	21
2.5. Creating repository instances	21
2.5.1. XML configuration	21
2.5.2. JavaConfig	22
2.5.3. Standalone usage	23
2.6. Custom implementations for Spring Data repositories	23
2.6.1. Customizing individual repositories	24
2.6.2. Customize the base repository	28
2.7. Publishing events from aggregate roots	29
2.8. Spring Data extensions	30
2.8.1. Querydsl Extension	30
2.8.2. Web support	31
2.8.3. Repository populators	38
2.8.4. Legacy web support	40
Reference Documentation	43
3. Solr Repositories	44

3.1. Introduction	44
3.1.1. Spring Namespace	44
3.1.2. Annotation based configuration	45
3.1.3. Solr Repositores using CDI	46
3.1.4. Transaction Support	47
3.2. Query methods	47
3.2.1. Query lookup strategies	48
3.2.2. Query creation	48
3.2.3. Using @Query Annotation	49
3.2.4. Using NamedQueries	49
3.3. Document Mapping	50
3.3.1. Mapping Solr Converter	50
4. Miscellaneous Solr Operation Support	53
4.1. Partial Updates	53
4.2. Projection	53
4.3. Faceting	53
4.3.1. Range Faceting	54
4.3.2. Pivot Faceting	56
4.4. Terms	56
4.5. Result Grouping / Field Collapsing	57
4.6. Field Stats	57
4.7. Filter Query	59
4.8. Time allowed for a search	60
4.9. Boost document Score	60
4.9.1. Index Time Boosts	60
4.10. Select Request Handler	61
4.11. Using Join	61
4.12. Highlighting	61
4.13. Spellchecking	62
4.13.1. Spellecheck Options	62
4.13.2. @Spellcheck	63
4.14. Using Functions	63
4.15. Realtime Get	64
4.16. Special Fields	64
4.16.1. @Score	64
4.17. Nested Documents	65
Appendix	67
Appendix A: Namespace reference	68
The <repositories /> element	68
Appendix B: Populators namespace reference	69
The <populator /> element	69

Appendix C: Repository query keywords.....	70
Supported query keywords.....	70
Appendix D: Repository query return types	71
Supported query return types	71

© 2012-2017 The original author(s).

NOTE

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

Preface

The Spring Data for Apache Solr project applies core Spring concepts to the development of solutions using the Apache Solr Search Engine. We provide a "template" as a high-level abstraction for storing and querying documents. You will notice similarities to the mongodb support in the Spring Framework.

Project Metadata

- Version Control - <https://github.com/spring-projects/spring-data-solr>
- Bugtacker - <https://jira.spring.io/browse/DATASOLR>
- Release repository - <https://repo.spring.io/libs-release>
- Milestone repository - <https://repo.spring.io/libs-milestone>
- Snapshot repository - <https://repo.spring.io/libs-snapshot>

Requirements

Requires Java 8 runtime and [Apache Solr](#) 6.6. Preferably the latest 6.6.x version.

```
<dependency>  
  <groupId>org.apache.solr</groupId>  
  <artifactId>solr-solrj</artifactId>  
  <version>${solr.version}</version>  
</dependency>
```


Chapter 1. New & Noteworthy

1.1. What's new in Spring Data for Apache Solr 2.1

- Autoselect Solr core using `SolrTemplate`.
- Assert upwards compatibility with Apache Solr 6 (incl. 6.3).
- Support for combined `Facet` and `Highlight` Query.
- Allow reading single valued multivalue fields into non collection property.
- Use native SolrJ schema api.

1.2. What's new in Spring Data for Apache Solr 2.0

- Upgrade to Apache Solr 5.
- Support `RequestMethod` when querying.

1.3. What's new in Spring Data for Apache Solr 1.5

- Support for `Range Faceting`.
- Automatically prefix/suffix map keys via `@Dynamic` (See: `dynamicMappedFieldValues` in `Mapping Solr Converter`).

1.4. What's new in Spring Data for Apache Solr 1.4

- Upgraded to recent Solr 4.10.x distribution (requires Java 7).
- Add support for `Realtime Get`.
- Get `Field Stats` (max, min, sum, count, mean, missing, stddev and distinct calculations).
- Use `@Score` to automatically add projection on document score (See: `Special Fields`).

Chapter 2. 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.

Spring Data repository documentation and your module

IMPORTANT

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.

2.1. 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 1. 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 2. 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 3. Derived Count Query

```
interface UserRepository extends CrudRepository<User, Long> {
    long countByLastname(String lastname);
}
```

Example 4. Derived Delete Query

```
interface UserRepository extends CrudRepository<User, Long> {
    long deleteByLastname(String lastname);
    List<User> removeByLastname(String lastname);
}
```

2.2. 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 explicitly 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.

2.3. 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`.

2.3.1. 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 5. 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.

2.3.2. 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 6. 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 7. 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 emailAddress); 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:


```
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.

2.3.3. 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 9. 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 10. 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 11. 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 12. 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 13. Annotation-driven configuration of base packages

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

2.4. 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.

2.4.1. 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.

2.4.2. 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 14. 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](#).

2.4.3. 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).

2.4.4. 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 15. 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.

2.4.5. 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 16. 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.

2.4.6. 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 17. 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 18. 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.

2.4.7. 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.

2.5. 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.

2.5.1. 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 19. 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 20. 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.

2.5.2. 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 21. 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.

2.5.3. 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 22. Standalone usage of repository factory

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

2.6. 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.

2.6.1. 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 23. Interface for custom repository functionality

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

Example 24. 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 25. 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 26. 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 27. 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 28. 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 29. 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 30. 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 31. Resolution of ambiguous implementations

```
package com.acme.impl.one;

class CustomizedUserRepositoryImpl implements CustomizedUserRepository {

    // Your custom implementation
}
```

```
package com.acme.impl.two;

@Component("specialCustomImpl")
class CustomizedUserRepositoryImpl implements CustomizedUserRepository {

    // Your custom implementation
}
```

If you annotate the `UserRepository` interface with `@Component("specialCustom")` the bean name plus `Impl` 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 32. Manual wiring of custom implementations

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

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

2.6.2. 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 33. 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 34. 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 35. Configuring a custom repository base class using `XML`

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

2.7. 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 36. 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.

2.8. 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.

2.8.1. 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 37. 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 38. 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);
```

2.8.2. 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 39. 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 40. 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 41. 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 42. 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 <code>property,property(,ASC DESC)</code> . Default sort direction is ascending. Use multiple <code>sort</code> parameters if you want to switch directions, e.g. <code>?sort=firstname&sort=lastname,asc</code> .

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 43. Using a `PagedResourcesAssembler` as controller method argument

```
@Controller
class PersonController {

    @Autowired PersonRepository repository;

    @RequestMapping(value = "/persons", method = RequestMethod.GET)
    ResponseEntity<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 from 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 44. 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 loopup 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>,
                                QuerydslBinderCustomizer<QUser> {

    ①
    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.

2.8.3. 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 45. 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 46. 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 47. 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>
```

2.8.4. 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 3. Solr Repositories

This chapter includes details of the Solr repository implementation.

3.1. Introduction

3.1.1. Spring Namespace

The Spring Data Solr module contains a custom namespace allowing definition of repository beans as well as elements for instantiating a `SolrClient`.

Using the `repositories` element looks up Spring Data repositories as described in [Creating repository instances](#).

Example 48. Setting up Solr repositories using 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:solr="http://www.springframework.org/schema/data/solr"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/solr
    http://www.springframework.org/schema/data/solr/spring-solr.xsd">

  <solr:repositories base-package="com.acme.repositories" />
</beans>
```

Using the `solr-server` or `embedded-solr-server` element registers an instance of `SolrClient` in the context.

Example 49. HttpSolrClient using 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:solr="http://www.springframework.org/schema/data/solr"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/solr
    http://www.springframework.org/schema/data/solr/spring-solr.xsd">

  <solr:solr-client id="solrClient" url="http://localhost:8983/solr" />
</beans>
```


Example 50. LBSolrClient using 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:solr="http://www.springframework.org/schema/data/solr"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/solr
    http://www.springframework.org/schema/data/solr/spring-solr.xsd">

  <solr:solr-client id="solrClient" url=
    "http://localhost:8983/solr,http://localhost:8984/solr" />
</beans>
```

Example 51. EmbeddedSolrServer using 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:solr="http://www.springframework.org/schema/data/solr"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/solr
    http://www.springframework.org/schema/data/solr/spring-solr.xsd">

  <solr:embedded-solr-server id="solrClient" solrHome="classpath:com/acme/solr" />
</beans>
```

3.1.2. Annotation based configuration

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

```
@Configuration
@EnableSolrRepositories
class ApplicationConfig {

    @Bean
    public SolrClient solrClient() {
        EmbeddedSolrServerFactory factory = new EmbeddedSolrServerFactory(
"classpath:com/acme/solr");
        return factory.getSolrServer();
    }

    @Bean
    public SolrOperations solrTemplate() {
        return new SolrTemplate(solrClient());
    }
}
```

The configuration above sets up an `EmbeddedSolrServer` which is used by the `SolrTemplate`. Spring Data Solr Repositories are activated using the `@EnableSolrRepositories` 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.

3.1.3. Solr Repositories using CDI

The Spring Data Solr repositories can also be set up using CDI functionality.

```
class SolrTemplateProducer {

    @Produces
    @ApplicationScoped
    public SolrOperations createSolrTemplate() {
        return new SolrTemplate(new EmbeddedSolrServerFactory("
classpath:com/acme/solr"));
    }
}

class ProductService {

    private ProductRepository repository;

    public Page<Product> findAvailableProductsByName(String name, Pageable pageable)
    {
        return repository.findByAvailableTrueAndNameStartingWith(name, pageable);
    }

    @Inject
    public void setRepository(ProductRepository repository) {
        this.repository = repository;
    }
}
```

3.1.4. Transaction Support

Solr supports transactions on server level means create, update, delete actions since the last commit/optimize/rollback are queued on the server and committed/optimized/rolled back at once. Spring Data Solr Repositories will participate in Spring Managed Transactions and commit/rollback changes on complete.

```
@Transactional
public Product save(Product product) {
    Product savedProduct = jpaRepository.save(product);
    solrRepository.save(savedProduct);
    return savedProduct;
}
```

3.2. Query methods

3.2.1. Query lookup strategies

The Solr module supports defining a query manually as String or have it being derived from the method name. NOTE: There is no QueryDSL Support present at this time.

Declared queries

Deriving the query from the method name is not always sufficient and/or may result in unreadable method names. In this case one might make either use of Solr named queries (see [Using NamedQueries](#)) or use the `@Query` annotation (see [Using @Query Annotation](#)).

3.2.2. Query creation

Generally the query creation mechanism for Solr works as described in [Query methods](#). Here's a short example of what a Solr query method translates into:

Example 54. Query creation from method names

```
public interface ProductRepository extends Repository<Product, String> {  
    List<Product> findByNameAndPopularity(String name, Integer popularity);  
}
```

The method name above will be translated into the following solr query

```
q=name:?0 AND popularity:?1
```

A list of supported keywords for Solr is shown below.

Table 2. Supported keywords inside method names

Keyword	Sample	Solr Query String
And	<code>findByNameAndPopularity</code>	<code>q=name:?0 AND popularity:?1</code>
Or	<code>findByNameOrPopularity</code>	<code>q=name:?0 OR popularity:?1</code>
Is	<code>findByName</code>	<code>q=name:?0</code>
Not	<code>findByNameNot</code>	<code>q=-name:?0</code>
IsNull	<code>findByNameIsNull</code>	<code>q=-name:[* TO *]</code>
NotNull	<code>findByNameNotNull</code>	<code>q=name:[* TO *]</code>
Between	<code>findByPopularityBetween</code>	<code>q=popularity:[?0 TO ?1]</code>
LessThan	<code>findByPopularityLessThan</code>	<code>q=popularity:[* TO ?0}</code>
LessThanEqual	<code>findByPopularityLessThanEqual</code>	<code>q=popularity:[* TO ?0]</code>
GreaterThan	<code>findByPopularityGreaterThan</code>	<code>q=popularity:{?0 TO *]</code>
GreaterThanEqual	<code>findByPopularityGreaterThanEqual</code>	<code>q=popularity:[?0 TO *]</code>
Before	<code>findByLastModifiedBefore</code>	<code>q=last_modified:[* TO ?0}</code>
After	<code>findByLastModifiedAfter</code>	<code>q=last_modified:{?0 TO *]</code>

Keyword	Sample	Solr Query String
Like	findByNameLike	q=name:?0*
NotLike	findByNameNotLike	q=-name:?0*
StartingWith	findByNameStartingWith	q=name:?0*
EndingWith	findByNameEndingWith	q=name:*?0
Containing	findByNameContaining	q=name:*?0*
Matches	findByNameMatches	q=name:?0
In	findByNameIn(Collection<String> names)	q=name:(?0...)
NotIn	findByNameNotIn(Collection<String> names)	q=-name:(?0...)
Within	findByStoreWithin(Point, Distance)	q={!geofilt pt=?0.latitude,?0.longitude sfield=store d=?1}
Near	findByStoreNear(Point, Distance)	q={!bbox pt=?0.latitude,?0.longitude sfield=store d=?1}
Near	findByStoreNear(Box)	q=store[?0.start.latitude,?0.start.longitude TO ?0.end.latitude,?0.end.longitude]
True	findByAvailableTrue	q=inStock:true
False	findByAvailableFalse	q=inStock:false
OrderBy	findByAvailableTrueOrderByName Desc	q=inStock:true&sort=name desc

NOTE

Collections types can be used along with 'Like', 'NotLike', 'StartingWith', 'EndingWith' and 'Containing'.

```
Page<Product> findByNameLike(Collection<String> name);
```

3.2.3. Using @Query Annotation

Using named queries ([Using NamedQueries](#)) 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 Solr `@Query` annotation.

Example 55. Declare query at the method using the @Query annotation.

```
public interface ProductRepository extends SolrRepository<Product, String> {
    @Query("inStock:?0")
    List<Product> findByAvailable(Boolean available);
}
```

3.2.4. Using NamedQueries

Named queries can be kept in a properties file and wired to the accrodg method. Please mind the naming convention described in [Query lookup strategies](#) or use `@Query` .

Example 56. Declare named query in properties file

```
Product.findByNamedQuery=popularity:?0  
Product.findByName=name:?0
```

```
public interface ProductRepository extends SolrCrudRepository<Product, String> {  
  
    List<Product> findByNamedQuery(Integer popularity);  
  
    @Query(name = "Product.findByName")  
    List<Product> findByAnnotatedNamedQuery(String name);  
  
}
```

3.3. Document Mapping

Though there is already support for Entity Mapping within SolrJ, Spring Data Solr ships with its own mapping mechanism shown in the following section. NOTE: `DocumentObjectBinder` has superior performance. Therefore usage is recommended if there is not need for custom type mapping. You can switch to `DocumentObjectBinder` by registering `SolrJConverter` within `SolrTemplate`.

3.3.1. Mapping Solr Converter

`MappingSolrConverter` allows you to register custom converters for your `SolrDocument` and `SolrInputDocument` as well as for other types nested within your beans. The Converter is not 100% compatible with `DocumentObjectBinder` and `@Indexed` has to be added with `readonly=true` to ignore fields from being written to solr.

Example 57. Sample Document Mapping

```
public class Product {
    @Field
    private String simpleProperty;

    @Field("somePropertyName")
    private String namedProperty;

    @Field
    private List<String> listOfValues;

    @Indexed(readonly = true)
    @Field("property_*")
    private List<String> ignoredFromWriting;

    @Field("mappedField_*")
    private Map<String, List<String>> mappedFieldValues;

    @Dynamic
    @Field("dynamicMappedField_*")
    private Map<String, String> dynamicMappedFieldValues;

    @Field
    private GeoLocation location;
}
```

Taking a look as the above `MappingSolrConverter` will do as follows:

Property	Write Mapping
simpleProperty	<code><field name="simpleProperty">value</field></code>
namedProperty	<code><field name="somePropertyName">value</field></code>
listOfValues	<code><field name="listOfValues">value 1</field> <field name="listOfValues">value 2</field> <field name="listOfValues">value 3</field></code>
ignoredFromWriting	<code>//not written to document</code>
mappedFieldValues	<code><field name="mapentry[0].key">mapentry[0].value[0]</field> <field name="mapentry[0].key">mapentry[0].value[1]</field> <field name="mapentry[1].key">mapentry[1].value[0]</field></code>
dynamicMappedFieldValues	<code><field name="'dynamicMappedField_' + mapentry[0].key">mapentry[0].value[0]</field> <field name="'dynamicMappedField_' + mapentry[0].key">mapentry[0].value[1]</field> <field name="'dynamicMappedField_' + mapentry[1].key">mapentry[1].value[0]</field></code>
location	<code><field name="location">48.362893,14.534437</field></code>

To register a custom converter one must add `CustomConversions` to `SolrTemplate` initializing it with own `Converter` implementation.

```
<bean id="solrConverter"
class="org.springframework.data.solr.core.convert.MappingSolrConverter">
  <constructor-arg>
    <bean
class="org.springframework.data.solr.core.mapping.SimpleSolrMappingContext" />
  </constructor-arg>
  <property name="customConversions" ref="customConversions" />
</bean>

<bean id="customConversions"
class="org.springframework.data.solr.core.convert.SolrCustomConversions">
  <constructor-arg>
    <list>
      <bean class="com.acme.MyBeanToSolrInputDocumentConverter" />
    </list>
  </constructor-arg>
</bean>

<bean id="solrTemplate" class="org.springframework.data.solr.core.SolrTemplate">
  <constructor-arg ref="solrClient" />
  <property name="solrConverter" ref="solrConverter" />
</bean>
```


Chapter 4. Miscellaneous Solr Operation Support

This chapter covers additional support for Solr operations (such as faceting) that cannot be directly accessed via the repository interface. It is recommended to add those operations as custom implementation as described in [Custom implementations for Spring Data repositories](#).

4.1. Partial Updates

PartialUpdates can be done using `PartialUpdate` which implements `Update`.

```
PartialUpdate update = new PartialUpdate("id", "123");
update.add("name", "updated-name");
solrTemplate.saveBean("collection-1", update);
```

4.2. Projection

Projections can be applied via `@Query` using the fields value.

```
@Query(fields = { "name", "id" })
List<ProductBean> findByNameStartingWith(String name);
```

4.3. Faceting

Faceting cannot be directly applied using the `SolrRepository` but the `SolrTemplate` holds support for this feature.

```
FacetQuery query = new SimpleFacetQuery(new Criteria(Criteria.WILDCARD).
expression(Criteria.WILDCARD))
    .setFacetOptions(new FacetOptions().addFacetOnField("name").setFacetLimit(5));
FacetPage<Product> page = solrTemplate.queryForFacetPage("collection-1", query,
Product.class);
```

Facets on fields and/or queries can also be defined using `@Facet`. Please mind that the result will be a `FacetPage`. NOTE: Using `@Facet` allows you to define place holders which will use your input parameter as value.

```
@Query(value = "*:.*")
@Facet(fields = { "name" }, limit = 5)
FacetPage<Product> findAllFacetOnName(Pageable page);
```

```
@Query(value = "popularity:?0")
@Facet(fields = { "name" }, limit = 5, prefix="?1")
FacetPage<Product> findByPopularityFacetOnName(int popularity, String prefix,
Pageable page);
```

Solr allows definition of facet parameters on a per field basis. In order to add special facet options to defined fields use `FieldWithFacetParameters`.

```
// produces: f.name.facet.prefix=spring
FacetOptions options = new FacetOptions();
options.addFacetOnField(new FieldWithFacetParameters("name").setPrefix("spring"));
```

4.3.1. Range Faceting

Range faceting queries may be done by configure required ranges on `FacetOptions`. A simple way to request ranges would be by creating a `FacetOption`, setting this options to a `FacetQuery` and query for a facet page through `SolrTemplate` as follows.

```

FacetOptions facetOptions = new FacetOptions()
    .addFacetByRange(
        new FieldWithNumericRangeParameters("price", 5, 20, 5)
            .setHardEnd(true)
            .setInclude(FacetRangeInclude.ALL)
    )
    .addFacetByRange(
        new FieldWithDateRangeParameters("release", new Date(1420070400), new Date(
946684800), "+1YEAR")
            .setInclude(FacetRangeInclude.ALL)
            .setOther(FacetRangeOther.BEFORE)
    );
facetOptions.setFacetMinCount(0);

Criteria criteria = new SimpleStringCriteria("*:");
SimpleFacetQuery facetQuery = new SimpleFacetQuery(criteria).setFacetOptions(
facetOptions);
FacetPage<ExampleSolrBean> statResultPage = solrTemplate.queryForFacetPage(
"collection-1", facetQuery, ExampleSolrBean.class);

```

There are two implementations of fields for facet range requests:

- **Numeric Facet Range** - used to perform range faceting over numeric fields. To request such range faceting an instance of the class `org.springframework.data.solr.core.query.FacetOptions.FieldWithNumericRangeParameters` can be used. Its instantiation requires a field name, a start value (number), end value (number) and gap (number);
- **Date Facet Range** - used to perform range faceting over date fields. To request such range faceting an instance of the class `org.springframework.data.solr.core.query.FacetOptions.FieldWithDateRangeParameters` can be used. Its instantiation requires a field name, a start value (date), end value (date) and gap (string). The gap for this kind of field can be defined using `org.apache.solr.util.DateMathParser` (i.e. `+6MONTHS+3DAYS/DAY`, that would mean 6 months and 3 days in the future from now, rounded down to nearest day).

Additionally the following properties can be configured for a field with range parameters (`org.springframework.data.solr.core.query.FacetOptions.FieldWithRangeParameters`):

- **Hard End** - `setHardEnd(Boolean)`, defines if the last range should be abruptly ended even if the end doesn't satisfies: $(start - end) \% gap = 0$;
- **Include** - `setInclude(org.apache.solr.common.params.FacetParams.FacetRangeInclude)`, defines how boundaries (lower and upper) shall be handled (exclusive or inclusive) on range facet requests;
- **Other** - `setOther(org.apache.solr.common.params.FacetParams.FacetRangeOther)`, defines the additional (other) counts for the range facet, i.e. count of documents that are before start of the range facet, end of range facet or even between start and end.

4.3.2. Pivot Faceting

Pivot faceting (Decision Tree) are also supported, and can be queried using `@Facet` annotation as follows:

```
public interface {

    @Facet(pivots = @Pivot({ "category", "dimension" }, pivotMinCount = 0))
    FacetPage<Product> findByTitle(String title, Pageable page);

    @Facet(pivots = @Pivot({ "category", "dimension" }))
    FacetPage<Product> findByDescription(String description, Pageable page);

}
```

Alternatively it can be queried using `SolrTemplate` as follows:

```
FacetQuery facetQuery = new SimpleFacetQuery(new SimpleStringCriteria("title:foo"
));
FacetOptions facetOptions = new FacetOptions();
facetOptions.setFacetMinCount(0);
facetOptions.addFacetOnPivot("category", "dimension");
facetQuery.setFacetOptions(facetOptions);
FacetPage<Product> facetResult = solrTemplate.queryForFacetPage("collection-1",
facetQuery, Product.class);
```

In order to retrieve the pivot results the method `getPivot` can be used as follows:

```
List<FacetPivotFieldEntry> pivot = facetResult.getPivot(new SimplePivotField(
"categories", "available"));
```

4.4. Terms

Terms Vector cannot directly be used within `SolrRepository` but can be applied via `SolrTemplate`. Please mind, that the result will be a `TermsPage`.

```
TermsQuery query = SimpleTermsQuery.queryBuilder().fields("name").build();
TermsPage page = solrTemplate.queryForTermsPage("collection-1", query);
```

4.5. Result Grouping / Field Collapsing

Result grouping cannot directly be used within `SolrRepository` but can be applied via `SolrTemplate`. Please mind, that the result will be a `GroupPage`.

```
Field field = new SimpleField("popularity");
Function func = ExistsFunction.exists("description");
Query query = new SimpleQuery("inStock:true");

SimpleQuery groupQuery = new SimpleQuery(new SimpleStringCriteria("*:*"));
GroupOptions groupOptions = new GroupOptions()
    .addGroupByField(field)
    .addGroupByFunction(func)
    .addGroupByQuery(query);
groupQuery.setGroupOptions(groupOptions);

GroupPage<Product> page = solrTemplate.queryForGroupPage("collection-1", query,
    Product.class);

GroupResult<Product> fieldGroup = page.getGroupResult(field);
GroupResult<Product> funcGroup = page.getGroupResult(func);
GroupResult<Product> queryGroup = page.getGroupResult(query);
```

4.6. Field Stats

Field stats are used to retrieve statistics (max, min, sum, count, mean, missing, stddev and distinct calculations) of given fields from Solr. It is possible by providing `StatsOptions` to your query and reading the `FieldStatsResult` from the returned `StatsPage`. This could be achieved for instance, using `SolrTemplate` as follows:

```
// simple field stats
StatsOptions statsOptions = new StatsOptions().addField("price");

// query
SimpleQuery statsQuery = new SimpleQuery("*:*");
statsQuery.setStatsOptions(statsOptions);
StatsPage<Product> statsPage = solrTemplate.queryForStatsPage("collection-1",
    statsQuery, Product.class);

// retrieving stats info
FieldStatsResult priceStatResult = statsPage.getFieldStatsResult("price");
Object max = priceStatResult.getMax();
Long missing = priceStatResult.getMissing();
```

The same result could be achieved annotating the repository method with `@Stats` as follows:

```
@Query("name:?0")
@Stats(value = { "price" })
StatsPage<Product> findByName(String name, Pageable page);
```

Distinct calculation and faceting are also supported:

```
// for distinct calculation
StatsOptions statsOptions = new StatsOptions()
    .addField("category")
    // for distinct calculation
    .setCalcDistinct(true)
    // for faceting
    .addFacet("availability");

// query
SimpleQuery statsQuery = new SimpleQuery("*:");
statsQuery.setStatsOptions(statsOptions);
StatsPage<Product> statsPage = solrTemplate.queryForStatsPage("collection-1",
statsQuery, Product.class);

// field stats
FieldStatsResult categoryStatResult = statResultPage.getFieldStatsResult("
category");

// retrieving distinct
List<Object> categoryValues = priceStatResult.getDistinctValues();
Long distinctCount = categoryStatResult.getDistinctCount();

// retrieving faceting
Map<String, StatsResult> availabilityFacetResult = categoryStatResult
.getFacetStatsResult("availability");
Long availableCount = availabilityFacetResult.get("true").getCount();
```

The annotated version of the sample above would be:

```
@Query("name:?0")
@Stats(value = "category", facets = { "availability" }, calcDistinct = true)
StatsPage<Product> findByName(String name);
```

In order to perform a selective faceting or selective distinct calculation, `@SelectiveStats` may be

used as follows:

```
// selective distinct faceting
...
Field facetField = getFacetField();
StatsOptions statsOptions = new StatsOptions()
    .addField("price")
    .addField("category").addSelectiveFacet("name").addSelectiveFacet(facetField);
...
// or annotating repository method as follows
...
@Stats(value = "price", selective = @SelectiveStats(field = "category", facets = {
    "name", "available" }))
...

// selective distinct calculation
...
StatsOptions statsOptions = new StatsOptions()
    .addField("price")
    .addField("category").setSelectiveCalcDistinct(true);
...
// or annotating repository method as follows
...
@Stats(value = "price", selective = @SelectiveStats(field = "category",
    calcDistinct = true))
...
```

4.7. Filter Query

Filter Queries improve query speed and do not influence document score. It is recommended to implement geospatial search as filter query. NOTE: Please note that in solr, unless otherwise specified, all units of distance are kilometers and points are in degrees of latitude,longitude.

```
Query query = new SimpleQuery(new Criteria("category").is(
    "supercalifragilisticexpialidocious"));
FilterQuery fq = new SimpleFilterQuery(new Criteria("store")
    .near(new Point(48.305478, 14.286699), new Distance(5)));
query.addFilterQuery(fq);
```

Simple filter queries can also be defined using `@Query` . NOTE: Using `@Query` allows you to define place holders which will use your input parameter as value.

```
@Query(value = ":*:*" , filters = { "inStock:true", "popularity:[* TO 3]" })
List<Product> findAllFilterAvailableTrueAndPopularityLessThanEqual3();
```

4.8. Time allowed for a search

It is possible to set the time allowed for a search to finish. This value only applies to the search and not to requests in general. Time is in milliseconds. Values less than or equal to zero implies no time restriction. Partial results may be returned, if there are any.

```
Query query = new SimpleQuery(new SimpleStringCriteria("field_1:value_1"));
// Allowing maximum of 100ms for this search
query.setTimeAllowed(100);
```

4.9. Boost document Score

Boost document score in case of matching criteria to influence result order. This can be done by either setting boost on `Criteria` or using `@Boost` for derived queries.

```
Page<Product> findByNameOrDescription(@Boost(2) String name, String description);
```

4.9.1. Index Time Boosts

Boosting documents score can be done on index time by using `@SolrDocument` annotation on classes (for Solr documents) and/or `@Indexed` on fields (for Solr fields).


```

import org.apache.solr.client.solrj.beans.Field;
import org.springframework.data.solr.repository.Boost;

@SolrDocument(boost = 0.8f)
public class MyEntity {

    @Id
    @Indexed
    private String id;

    @Indexed(boost = 1.0f)
    private String name;

    // setters and getters ...

}

```

4.10. Select Request Handler

Select the request handler via `qt` Parameter directly in `Query` or add `@Query` to your method signature.

```

@Query(requestHandler = "/instock")
Page<Product> findByNameOrDescription(String name, String description);

```

4.11. Using Join

Join attributes within one solr core by defining `Join` attribute of `Query`. NOTE: Join is not available prior to solr 4.x.

```

SimpleQuery query = new SimpleQuery(new SimpleStringCriteria("text:ipod"));
query.setJoin(Join.from("manu_id_s").to("id"));

```

4.12. Highlighting

To highlight matches in search result add `HighlightOptions` to the `SimpleHighlightQuery`. Providing `HighlightOptions` without any further attributes will highlight apply highlighting on all fields within a `SolrDocument`. NOTE: Field specific highlight parameters can be set by adding `FieldWithHighlightParameters` to `HighlightOptions`.

```
SimpleHighlightQuery query = new SimpleHighlightQuery(new SimpleStringCriteria(
    "name:with"));
query.setHighlightOptions(new HighlightOptions());
HighlightPage<Product> page = solrTemplate.queryForHighlightPage("collection-1",
    query, Product.class);
```

Not all parameters are available via setters/getters but can be added directly.

```
SimpleHighlightQuery query = new SimpleHighlightQuery(new SimpleStringCriteria(
    "name:with"));
query.setHighlightOptions(new HighlightOptions().addHighlightParameter(
    "hl.bs.country", "at"));
```

In order to apply Highlighting to derived queries use `@Highlight`. If no `fields` are defined highlighting will be applied on all fields.

```
@Highlight(prefix = "<b>", postfix = "</b>")
HighlightPage<Product> findByName(String name, Pageable page);
```

4.13. Spellchecking

Spellchecking offers search term suggestions based on the actual query. Please see the [Solr Reference](#) for more details.

4.13.1. Spellecheck Options

Spellcheck query parameters are added to request when `SpellcheckOptions` are set.

```

SimpleQuery q = new SimpleQuery("name:gren");
q.setSpellcheckOptions(SpellcheckOptions.spellcheck()           ①
    .dictionaries("dict1", "dict2")                             ②
    .count(5)                                                   ③
    .extendedResults());                                       ④
q.setRequestHandler("/spell");                                  ⑤

SpellcheckedPage<Product> found = template.query(q, Product.class); ⑥

```

- ① Enable spellcheck by setting `SpellcheckOptions`. Sets `spellcheck=on` request parameter.
- ② Set up the dictionaries to use for lookup.
- ③ Set the max number of suggestions to return.
- ④ Enable extended results including term frequency etc.
- ⑤ Set the request handler capable of processing suggestions.
- ⑥ Execute the query.

4.13.2. @Spellcheck

The `@Spellcheck` annotations allows usage of the spellcheck feature on `Repository` level.

```

public interface ProductRepository extends Repository<Product, String> {

    @Query(requestHandler = "/spell")
    @Spellcheck(dictionaries = { "dict1", "dic2" }, count=5, extendedResults = true)
    SpellcheckedPage<Product> findByName(String name, Pageable page);

}

```

4.14. Using Functions

Solr supports several functional expressions within queries. Followig functions are supported out of the box. Custom functions can be added by implementing `Function`

Table 3. Functions

Class	Solr Function
<code>CurrencyFunction</code>	<code>currency(field_name,[CODE])</code>
<code>DefaultValueFunction</code>	<code>def(field function,defaultValue)</code>
<code>DistanceFunction</code>	<code>dist(power, pointA, pointB)</code>
<code>DivideFunction</code>	<code>div(x,y)</code>
<code>ExistsFunction</code>	<code>exists(field function)</code>
<code>GeoDistanceFunction</code>	<code>geodist(sfield, latitude, longitude)</code>

Class	Solr Function
GeoHashFunction	geohash(latitude, longitude)
IfFunction	if(value field function,trueValue,falseValue)
MaxFunction	max(field function,value)
NotFunction	not(field function)
ProductFunction	product(x,y,...)
QueryFunction	query(x)
TermFrequencyFunction	termfreq(field,term)

```
SimpleQuery query = new SimpleQuery(new SimpleStringCriteria("text:ipod"));
query.addFilterQuery(new FilterQuery(Criteria.where(QueryFunction.query("
name:sol*"))));
```

4.15. Realtime Get

The realtime get allows retrieval of the latest version of any document using the unique-key, without the need to reopen searchers.

NOTE | realtime get relies on the update log feature.

Example 58. Realtime get

```
Optional<Product> product = solrTemplate.getById("collection-1", "123", Product
.class);
```

Multiple documents can be retrieved by providing a collection of ids as follows:

Example 59. Realtime multi-get

```
Collection<String> ids = Arrays.asList("123", "134");
Collection<Product> products = solrTemplate.getByIds("collection-1", ids, Product
.class);
```

4.16. Special Fields

4.16.1. @Score

In order to load score information of a query result, a field annotated with `@Score` annotation could be added, indicating the property holding the documents score.

NOTE

The score property needs to be numerical and can only appear once per document.

```
public class MyEntity {  
  
    @Id  
    private String id;  
  
    @Score  
    private Float score;  
  
    // setters and getters ...  
  
}
```

4.17. Nested Documents

Nested Documents provides the ability to add documents inside of other documents in a parent/child relationship.

The nested documents need to be indexed along with the parent one and cannot be updated individually. Though nested documents will appear as individual ones in the index. Resolving the parent child relation is done at query query time.

To indicate a property should be treated as nested object it has to be annotated with either `@o.a.s.c.solrj.beans.Field(child=true)` or `@o.s.d.s.core.mapping.ChildDocument`.

```

public class Book {

    @Id String id;
    @Indexed("type_s") String type;
    @Indexed("title_t") String title;
    @Indexed("author_s") String author;
    @Indexed("publisher_s") String publisher;

    @ChildDocument List<Review> reviews;    ①

    // setters and getters ...

}

public class Review {

    @Id String id;    ②
    @Indexed("type_s") String type;
    @Indexed("review_dt") Date date;
    @Indexed("stars_i") int stars;
    @Indexed("author_s") String author;
    @Indexed("comment_t") String comment;

}

```

- ① Multiple child documents can be associated with a parent one, or just use the domain type to store a single relation ship.
- ② Note that the nested document also needs to have a unique `id` assigned.

Assuming `Book#type` is `book`, and `Review#type` resolves to `review` retrieving `Book` with its child relations `reviews` can be done by altering the `fl` query parameter.

```

Query query = new SimpleQuery(where("id").is("theWayOfKings"));
query.addProjectionOnField(new SimpleField("*"));
query.addProjectionOnField(new SimpleField("[child parentFilter=type_s:book]"));
①

return solrTemplate.queryForObject("books", query, Book.class);

```

- ① The parent filter always defines the complete set of parent documents in the index, not the one for a single document.

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 4. Attributes

Name	Description
<code>base-package</code>	Defines the package to be used to be scanned for repository interfaces extending <code>*Repository</code> (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.
<code>repository-impl-postfix</code>	Defines the postfix to autodetect custom repository implementations. Classes whose names end with the configured postfix will be considered as candidates. Defaults to <code>Impl</code> .
<code>query-lookup-strategy</code>	Determines the strategy to be used to create finder queries. See Query lookup strategies for details. Defaults to <code>create-if-not-found</code> .
<code>named-queries-location</code>	Defines the location to look for a Properties file containing externally defined queries.
<code>consider-nested-repositories</code>	Controls whether nested repository interface definitions should be considered. Defaults to <code>false</code> .

Appendix B: Populators namespace reference

The <populator /> element

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

Table 5. Attributes

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

Appendix C: Repository query keywords

Supported query keywords

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

Table 6. Query keywords

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

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

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