Spring Data Solr

Christoph Strobl, Oliver Gierke, Mark Pollack, Thomas Risberg

Version 1.4.0.RELEASE
2015-03-23

Table of Contents

Preface

The Spring Data 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.

Chapter 1. Project Metadata

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

Chapter 2. Requirements

Requires Apache Solr 3.6 and above or optional dependency

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

NOTE If you tend to use the Embedded Version of Solr Server 4.x you will also have to add a version of servlet-api and check your <lockType> as well as <unlockOnStartup> settings.

Chapter 3. New & Noteworthy

3.1. What's new in Spring Data 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 4. Working with Spring Data Repositories

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

IMPORTANT

Spring Data repository documentation and your module

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

4.1. 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>

    T findOne(ID primaryKey);       <2>

    Iterable<T> findAll();          <3>

    Long count();                   <4>

    void delete(T entity);          <5>

    boolean exists(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

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

  Long countByLastname(String lastname);
}
```

Example 4. Derived Delete Query

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

  Long deleteByLastname(String lastname);

  List<User> removeByLastname(String lastname);

}
```

4.2. Query methods

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

1. Declare an interface extending Repository or one of its subinterfaces and type it to the domain class and ID type that it will handle.

```
interface PersonRepository extends Repository<User, Long> {    }
```

2. Declare query methods on the interface.

```
interface PersonRepository extends Repository<User, Long> {
  List<Person> findByLastname(String lastname);
}
```

3. Set up Spring to create proxy instances for those interfaces. Either via JavaConfig:

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

@EnableJpaRepositories
class Config {}
```

   or via XML configuration:

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

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

</beans>
```

   The JPA namespace is used in this example. If you are using the repository abstraction for any other store, you need to change this to the appropriate namespace declaration of your store module which should be exchanging `jpa` in favor of, for example, `mongodb`. Also, note that the JavaConfig variant doesn't configure a package explictly as the package of the annotated class is used by default. To customize the package to scan

4. Get the repository instance injected and use it.

```
public class SomeClient {

  @Autowired
  private PersonRepository repository;

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

The sections that follow explain each step in detail.

4.3. 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`.

4.3.1. 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> {

  T findOne(ID id);

  T save(T 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 `findOne( )` 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.

4.4. 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 an strategy that decides what actual query is created. Let's have a look at the available options.

4.4.1. 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.

4.4.2. Query creation

The query builder mechanism built into Spring Data repository infrastructure is useful for building constraining queries over entities of the repository. The mechanism strips the prefixes `find…By`, `read…By`, `query…By`, `count…By`, and `get…By` from the method and starts parsing the rest of it. The introducing clause can contain further expressions such as a `Distinct` to set a distinct flag on the query

to be created. However, the first `By` acts as delimiter to indicate the start of the actual criteria. At a very basic level you can define conditions on entity properties and concatenate them with `And` and `Or`.

Example 6. Query creation from method names

```
public 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.

4.4.3. Property expressions

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

If your property names contain underscores (e.g. `first_name`) you can escape the underscore in the method name with a second underscore. For a `first_name` property the query method would have to be named `findByFirst__name( )`.

4.4.4. 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 7. 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 thought 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.

4.4.5. 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 8. 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.

4.4.6. Streaming query results

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

Example 9. 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 10. 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.

4.5. 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.

4.5.1. 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 11. 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 12. 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.

4.5.2. 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. [JavaConfig in the Spring reference documentation]

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

Example 13. Sample annotation based repository configuration

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

  @Bean
  public 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.

4.5.3. 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 14. Standalone usage of repository factory

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

4.6. Custom implementations for Spring Data repositories

Often it is necessary to provide a custom implementation for a few repository methods. Spring Data repositories easily allow you to provide custom repository code and integrate it with generic CRUD abstraction and query method functionality.

4.6.1. Adding custom behavior to single repositories

To enrich a repository with custom functionality you first define an interface and an implementation for the custom functionality. Use the repository interface you provided to extend the custom interface.

Example 15. Interface for custom repository functionality

```
interface UserRepositoryCustom {
  public void someCustomMethod(User user);
}
```

Example 16. Implementation of custom repository functionality

```
class UserRepositoryImpl implements UserRepositoryCustom {

  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 core repository interface (see below).

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 JdbTemplate, take part in aspects, and so on.

Example 17. Changes to the your basic repository interface

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

  // Declare query methods here
}
```

Let your standard repository interface extend the custom one. Doing so combines the CRUD and custom functionality and makes it available to clients.

Configuration

If you use namespace configuration, the repository infrastructure tries to autodetect custom implementations 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 repository interface name. This postfix defaults to `Impl`.

Example 18. 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.UserRepositoryImpl` to act as custom repository implementation, whereas the second example will try to lookup

`com.acme.repository.UserRepositoryFooBar`.

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 custom implementation 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 19. Manual wiring of custom implementations

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

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

4.6.2. Adding custom behavior to all repositories

The preceding approach is not feasible when you want to add a single method to all your repository interfaces.

1. To add custom behavior to all repositories, you first add an intermediate interface to declare the shared behavior.

   Example 20. An interface declaring custom shared behavior

   ```
   @NoRepositoryBean
   public interface MyRepository<T, ID extends Serializable>
     extends PagingAndSortingRepository<T, ID> {

     void sharedCustomMethod(ID id);
   }
   ```

2. Now your individual repository interfaces will extend this intermediate interface instead of the `Repository` interface to include the functionality declared.

3. Next, create an implementation of the intermediate interface that extends the persistence technology-specific repository base class. This class will then act as a custom base class for the repository proxies.

Example 21. Custom repository base class

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

  private final EntityManager entityManager;

  public MyRepositoryImpl(Class<T> domainClass, EntityManager entityManager) {
    super(domainClass, entityManager);

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

  public void sharedCustomMethod(ID id) {
    // implementation goes here
  }
}
```

The default behavior of the Spring `<repositories />` namespace is to provide an implementation for all interfaces that fall under the `base-package`. This means that if left in its current state, an implementation instance of `MyRepository` will be created by Spring. This is of course not desired as it is just supposed to act as an intermediary between `Repository` and the actual repository interfaces you want to define for each entity. To exclude an interface that extends `Repository` from being instantiated as a repository instance, you can either annotate it with `@NoRepositoryBean` (as seen above) or move it outside of the configured `base-package`.

4. Then create a custom repository factory to replace the default `RepositoryFactoryBean` that will in turn produce a custom `RepositoryFactory`. The new repository factory will then provide your `MyRepositoryImpl` as the implementation of any interfaces that extend the `Repository` interface, replacing the `SimpleJpaRepository` implementation you just extended.

Example 22. Custom repository factory bean

```
public class MyRepositoryFactoryBean<R extends JpaRepository<T, I>, T,
  I extends Serializable> extends JpaRepositoryFactoryBean<R, T, I> {

  protected RepositoryFactorySupport createRepositoryFactory(EntityManager em) {
    return new MyRepositoryFactory(em);
  }

  private static class MyRepositoryFactory<T, I extends Serializable>
    extends JpaRepositoryFactory {

    private final EntityManager em;

    public MyRepositoryFactory(EntityManager em) {

      super(em);
      this.em = em;
    }

    protected Object getTargetRepository(RepositoryMetadata metadata) {
      return new MyRepositoryImpl<T, I>((Class<T>) metadata.getDomainClass(), em);
    }

    protected Class<?> getRepositoryBaseClass(RepositoryMetadata metadata) {
      return MyRepositoryImpl.class;
    }
  }
}
```

5. Finally, either declare beans of the custom factory directly or use the `factory-class` attribute of the Spring namespace or `@Enable` annotation to instruct the repository infrastructure to use your custom factory implementation.

Example 23. Using the custom factory with the namespace

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

Example 24. Using the custom factory with the `@Enable` annotation

```
@EnableJpaRepositories(factoryClass = "com.acme.MyRepositoryFactoryBean")
class Config {}
```

4.7. Spring Data extensions

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

4.7.1. 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 [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 25. 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 26. 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 27. A Spring MVC controller using domain types in method signatures

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

  @RequestMapping("/{id}")
  public 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 `findOne( )` 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 28. Using Pageable as controller method argument

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

  @Autowired UserRepository repository;

  @RequestMapping
  public 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.
`size`	Size of the page you want to retrieve.
`sort`	Properties that should be sorted by in the format `property,property(,ASC|DESC)`. Default sort direction is ascending. Use multiple `sort` parameters if you want to switch directions, e.g. `?sort=firstname&sort=lastname,asc`.

To customize this behavior 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:

```
public 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 `@PageableDefaults` annotation on the `Pageable` parameter.

Hypermedia support for Pageables

Spring HATEOAS ships with a representation model class `PagedResources` that allows enrichting 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 29. Using a PagedResourcesAssembler as controller method argument

```
@Controller
class PersonController {

  @Autowired PersonRepository repository;

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

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

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

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

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

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

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

4.7.2. 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 30. 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 31. 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:jackson-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 32. 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>
```

4.7.3. 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")
public class UserController {

  private final UserRepository userRepository;

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

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

    // Do null check for id
    User user = userRepository.findOne(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 `findOne( )` 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")
public class UserController {

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

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

ConversionServiceIn Spring 3.0 and later the `PropertyEditor` support is superseded by a new conversion infrastructure that eliminates the drawbacks of `PropertyEditors` and uses a stateless X to Y conversion approach. Spring Data now ships with a `DomainClassConverter` that mimics the behavior of `DomainClassPropertyEditorRegistrar`. To configure, simply declare a bean instance and pipe the `ConversionService` being used into its constructor:

```
<mvc:annotation-driven conversion-service="conversionService" />

<bean class="org.springframework.data.repository.support.DomainClassConverter">
  <constructor-arg ref="conversionService" />
</bean>
```

If you are using JavaConfig, you can simply extend Spring MVC's `WebMvcConfigurationSupport` and hand the `FormatingConversionService` that the configuration superclass provides into the `DomainClassConverter` instance you create.

```
class WebConfiguration extends WebMvcConfigurationSupport {

  // Other configuration omitted

  @Bean
  public DomainClassConverter<?> domainClassConverter() {
    return new DomainClassConverter<FormattingConversionService>(mvcConversionService());
  }
}
```

Web pagination

When working with pagination in the web layer you usually have to write a lot of boilerplate code yourself to extract the necessary metadata from the request. The less desirable approach shown in the example below requires the method to contain an `HttpServletRequest` parameter that has to be parsed manually. This example also omits appropriate failure handling, which would make the code even more verbose.

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

  // DI code omitted

  @RequestMapping
  public String showUsers(Model model, HttpServletRequest request) {

    int page = Integer.parseInt(request.getParameter("page"));
    int pageSize = Integer.parseInt(request.getParameter("pageSize"));

    Pageable pageable = new PageRequest(page, pageSize);

    model.addAttribute("users", userService.getUsers(pageable));
    return "users";
  }
}
```

The bottom line is that the controller should not have to handle the functionality of extracting pagination information from the request. So Spring Data ships with a `PageableHandlerMethodArgumentResolver` that will do the work for you. The Spring MVC JavaConfig support exposes a `WebMvcConfigurationSupport` helper class to customize the configuration as follows:

```
@Configuration
public class WebConfig extends WebMvcConfigurationSupport {

  @Override
  protected void addArgumentResolvers(List<HandlerMethodArgumentResolver>
argumentResolvers) {
    argumentResolvers.add(new PageableHandlerMethodArgumentResolver());
  }
}
```

If you're stuck with XML configuration you can register the resolver as follows:

```
<bean class="  .web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">
  <property name="customArgumentResolvers">
    <list>
      <bean class="org.springframework.data.web.PageableHandlerMethodArgumentResolver" />
    </list>
  </property>
</bean>
```

Once you've configured the resolver with Spring MVC it allows you to simplify controllers down to something like this:

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

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

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

The `PageableArgumentResolver` automatically resolves request parameters to build a `PageRequest` instance. By default it expects the following structure for the request parameters.

Table 2. Request parameters evaluated by PageableHandlerMethodArgumentResolver

`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`	A collection of sort directives in the format `($propertyname,)[asc|desc]?`.

To retrieve the third page with a maximum page size of 100 with the data sorted by the email property in ascending order use the following url parameter:

```
?page=2&size=100&sort=email,asc
```

To sort the data by multiple properties in different sort order use the following URL parameter:

```
?sort=foo,asc&sort=bar,desc
```

In case you need multiple `Pageable` 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:

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

you have to populate `foo_page` and `bar_page` and the related subproperties.

Configuring a global default on bean declaration the `PageableArgumentResolver` will use a `PageRequest` with the first page and a page size of 10 by default. It will use that value if it cannot resolve a `PageRequest` from the request (because of missing parameters, for example). You can configure a global default on the bean declaration directly. If you might need controller method specific defaults for the `Pageable`, annotate the method parameter with `@PageableDefaults` and specify page (through `pageNumber`), page size (through `value`), `sort` (list of properties to sort by), and `sortDir` (the direction to sort by) as annotation attributes:

```
public String showUsers(Model model,
  @PageableDefaults(pageNumber = 0, value = 30) Pageable pageable) {    }
```

Reference Documentation

Chapter 5. Solr Repositories

This chapter includes details of the Solr repository implementation.

5.1. Introduction

5.1.1. Spring Namespace

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

Using the `repositories` element looks up Spring Data repositories as described in Creating repository instances .

Example 33. 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-1.0.xsd">

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

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

Example 34. HttpSolrServer 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-1.0.xsd">

  <solr:solr-server id="solrServer" url="http://locahost:8983/solr" />
</beans>
```

Example 35. LBSolrServer 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-1.0.xsd">

  <solr:solr-server id="solrServer" url=
"http://locahost:8983/solr,http://localhost:8984/solr" />
</beans>
```

Example 36. 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-1.0.xsd">

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

5.1.2. Annotation based configuration

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

Example 37. Spring Data Solr repositories using JavaConfig

```
@Configuration
@EnableSolrRepositories
class ApplicationConfig {

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

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

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.

5.1.3. Multicore Support

Solr handles different collections within one core. Use `MulticoreSolrServerFactory` to create separate `SolrServer` for each core.

Example 38. Multicore Configuration

```
@Configuration
@EnableSolrRepositories(multicoreSupport = true)
class ApplicationConfig {

  private static final String PROPERTY_NAME_SOLR_SERVER_URL = "solr.host";

  @Resource
  private Environment environment;

  @Bean
  public SolrServer solrServer() {
    return new HttpSolrServer(environment.getRequiredProperty
(PROPERTY_NAME_SOLR_SERVER_URL));
  }

}
```

5.1.4. Solr Repositores using CDI

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

Example 39. Spring Data Solr repositories using JavaConfig

```
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;
  }
}
```

5.1.5. Transaction Support

Solr supports transactions on server level means create, updaet, 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;
}
```

5.2. Query methods

5.2.1. 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).

5.2.2. 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 40. 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 3. Supported keywords inside method names

Keyword	Sample	Solr Query String
`And`	`findByNameAndPopularity`	`q=name:?0 AND popularity:?1`
`Or`	`findByNameOrPopularity`	`q=name:?0 OR popularity:?1`
`Is`	`findByName`	`q=name:?0`
`Not`	`findByNameNot`	`q=-name:?0`
`IsNull`	`findByNameIsNull`	`q=-name:[* TO *]`
`IsNotNull`	`findByNameIsNotNull`	`q=name:[* TO *]`
`Between`	`findByPopularityBetween`	`q=popularity:[?0 TO ?1]`

Keyword	Sample	Solr Query String
LessThan	findByPopularityLessThan	q=popularity:[* TO ?0}
LessThanEqual	findByPopularityLessThanEqual	q=popularity:[* TO ?0]
GreaterThan	findByPopularityGreaterThan	q=popularity:{?0 TO *]
GreaterThanEqual	findByPopularityGreaterThanEqual	q=popularity:[?0 TO *]
Before	findByLastModifiedBefore	q=last_modified:[* TO ?0}
After	findByLastModifiedAfter	q=last_modified:{?0 TO *]
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	findByAvailableTrueOrderByNameDesc	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);
```

5.2.3. 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 41. Declare query at the method using the `@Query` annotation.

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

5.2.4. Using NamedQueries

Named queries can be kept in a properties file and wired to the accroding method. Please mind the naming convention described in Query lookup strategies or use `@Query` .

Example 42. 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);

}
```

5.3. 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.

5.3.1. 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% compartible with `DocumentObjectBinder` and `@Indexed` has to be added with `readonly=true` to ignore fields

from beeing written to solr.

Example 43. Sample Document Mapping

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

  @Field("somePropertyName")
  private String namedPropery;

  @Field
  private List<String> listOfValues;

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

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

  @Field
  private GeoLocation location;

}
```

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

Property	**Write Mapping**
simpleProperty	`<field name="simpleProperty">value</field>`
namedPropery	`<field name="somePropertyName">value</field>`
listOfValues	`<field name="listOfValues">value 1</field> <field name="listOfValues">value 2</field> <field name="listOfValues">value 3</field>`
ignoredFromWriting	`//not written to document`
mappedFieldValues	`<field name="mapentry[0].key">mapentry[0].value[0]</field> <field name="mapentry[0].key">mapentry[0].value[2]</field> <field name="mapentry[1].key">mapentry[1].value[0]</field>`
location	`<field name="location">48.362893,14.534437</field>`

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.CustomConversions">
<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="solrServer" />
<property name="solrConverter" ref="solrConverter" />
</bean>
```

Chapter 6. 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 .

6.1. Partial Updates

PartialUpdates can be done using `PartialUpdate` which implements `Update` . NOTE: Partial updates require Solr 4.x. With Solr 4.0.0 it is not possible to update mulitvalue fields.

NOTE	With Solr 4.1.0 you have to take care on parameter order when setting null values. Order parameters with nulls last.

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

6.2. Projection

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

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

6.3. 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(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"));
```

6.3.1. 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(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"));
```

6.4. 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(query);
```

6.5. 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(query, Product.class);

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

6.6. 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(statsQuery, Product
.class);

// retrieving stats info
FieldStatsResult priceStatResult = statResultPage.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" })
Stats<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(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))
...
```

6.7. 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();
```

6.8. Time allowed for a search

It it 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);
```

6.9. 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);
```

6.9.1. 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 ...

}
```

6.10. 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);
```

6.11. 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"));
```

6.12. 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(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 aplied on all fields.

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

6.13. 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 4. Functions

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

Class	Solr Function
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*")
)));
```

6.14. 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 44. Realtime get

```
Product product = solrTemplate.getById("123", Product.class);
```

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

Example 45. Realtime multi-get

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

6.15. Special Fields

6.15.1. @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 ...

}
```

Appendix

Appendix A: Namespace reference

The <repositories /> element

The `<repositories />` element triggers the setup of the Spring Data repository infrastructure. The most important attribute is `base-package` which defines the package to scan for Spring Data repository interfaces. [see XML configuration]

Table 5. Attributes

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

Appendix B: Populators namespace reference

The <populator /> element

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

Table 6. Attributes

Name	Description
`locations`	Where to find the files to read the objects from the repository shall be populated with.

Appendix C: Repository query keywords

Supported query keywords

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

Table 7. Query keywords

Logical keyword	Keyword expressions
`AND`	`And`
`OR`	`Or`
`AFTER`	`After`, `IsAfter`
`BEFORE`	`Before`, `IsBefore`
`CONTAINING`	`Containing`, `IsContaining`, `Contains`
`BETWEEN`	`Between`, `IsBetween`
`ENDING_WITH`	`EndingWith`, `IsEndingWith`, `EndsWith`
`EXISTS`	`Exists`
`FALSE`	`False`, `IsFalse`
`GREATER_THAN`	`GreaterThan`, `IsGreaterThan`
`GREATER_THAN_EQUALS`	`GreaterThanEqual`, `IsGreaterThanEqual`
`IN`	`In`, `IsIn`
`IS`	`Is`, `Equals`, (or no keyword)
`IS_NOT_NULL`	`NotNull`, `IsNotNull`
`IS_NULL`	`Null`, `IsNull`
`LESS_THAN`	`LessThan`, `IsLessThan`
`LESS_THAN_EQUAL`	`LessThanEqual`, `IsLessThanEqual`
`LIKE`	`Like`, `IsLike`
`NEAR`	`Near`, `IsNear`
`NOT`	`Not`, `IsNot`
`NOT_IN`	`NotIn`, `IsNotIn`
`NOT_LIKE`	`NotLike`, `IsNotLike`

Logical keyword	Keyword expressions
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 8. Query return types

Return type	Description
`void`	Denotes no return value.
Primitives	Java primitives.
Wrapper types	Java wrapper types.
`T`	An unique entity. Expects the query method to return one result at most. In case no result is found `null` is returned. More than one result will trigger an `IncorrectResultSizeDataAccessException`.
`Iterator<T>`	An `Iterator`.
`Collection<T>`	A `Collection`.
`List<T>`	A `List`.
`Optional<T>`	A Java 8 or Guava `Optional`. Expects the query method to return one result at most. In case no result is found `Optional.empty()`/`Optional.absent()` is returned. More than one result will trigger an `IncorrectResultSizeDataAccessException`.
`Stream<T>`	A Java 8 `Stream`.
`Slice`	A sized chunk of data with information whether there is more data available. Requires a `Pageable` method parameter.
`Page<T>`	A Slice with additional information, e.g. the total number of results. Requires a `Pageable` method parameter.
`GeoResult<T>`	A result entry with additional information, e.g. distance to a reference location.
`GeoResults<T>`	A list of GeoResult<T> with additional information, e.g. average distance to a reference location.
`GeoPage<T>`	A Page with GeoResult<T>, e.g. average distance to a reference location.