Spring Data Cassandra - Reference Documentation

David Webb, Matthew Adams

Version 1.4.7.RELEASE, 2017-01-26

Table of Contents

Preface

The Spring Data Cassandra project applies core Spring concepts to the development of solutions using the Cassandra Columnar data store. We provide a "template" as a high-level abstraction for storing and querying documents. You will notice similarities to the JDBC support in the Spring Framework.

Chapter 1. Project Metadata

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

Introduction

This document is the reference guide for Spring Data - Cassandra Support. It explains Cassandra module concepts and semantics and the syntax for various stores namespaces.

This section provides some basic introduction to Spring and the Cassandra database. The rest of the document refers only to Spring Data Cassandra features and assumes the user is familiar with Cassandra as well as Spring concepts.

Knowing Spring

Spring Data uses Spring framework's core functionality, such as the IoC container, type conversion system, expression language, JMX integration, and portable DAO exception hierarchy. While it is not important to know the Spring APIs, understanding the concepts behind them is. At a minimum, the idea behind IoC should be familiar for whatever IoC container you choose to use.

The core functionality of the Cassandra support can be used directly, with no need to invoke the IoC services of the Spring Container. This is much like `JdbcTemplate` which can be used 'standalone' without any other services of the Spring container. To leverage all the features of Spring Data Cassandra, such as the repository support, you will need to configure some parts of the library using Spring.

To learn more about Spring, you can refer to the comprehensive (and sometimes disarming) documentation that explains in detail the Spring Framework. There are a lot of articles, blog entries and books on the matter - take a look at the Spring framework home page for more information.

Knowing NoSQL and Cassandra

NoSQL stores have taken the storage world by storm. It is a vast domain with a plethora of solutions, terms and patterns (to make things worth even the term itself has multiple meanings). While some of the principles are common, it is crucial that the user is familiar to some degree with the Cassandra Columnar NoSQL Datastore supported by DATACASS. The best way to get acquainted to this solutions is to read their documentation and follow their examples - it usually doesn't take more then 5-10 minutes to go through them and if you are coming from an RDMBS-only background many times these exercises can be an eye opener.

The jumping off ground for learning about Cassandra is cassandra.apache.org/. Here is a list of

other useful resources.

- The Planet Cassandra site has many valuable resources for Cassandra best practices.

The DataStax site offers commercial support and many resources.

Chapter 2. Requirements

Spring Data Cassandra 1.x binaries requires JDK level 6.0 and above, and Spring Framework 3.2.x and above.

Currently we support Cassandra 2.X using the DataStax Java Driver (2.0.X)

Chapter 3. Additional Help Resources

Learning a new framework is not always straight forward. In this section, we try to provide what we think is an easy to follow guide for starting with Spring Data Cassandra module. However, if you encounter issues or you are just looking for an advice, feel free to use one of the links below:

3.1. Support

There are a few support options available:

3.1.1. Questions & Answers

Developers post questions and answers on Stack Overflow. The two key tags to search for related answers to this project are:

- spring-data
- spring-data-cassandra

3.1.2. Professional Support

Professional, from-the-source support, with guaranteed response time, is available from Pivotal Support.

3.2. Following Development

For information on the Spring Data Cassandra source code repository, nightly builds and snapshot artifacts please see the Spring Data Cassandra homepage.

To follow developer activity look for the mailing list information on the Spring Data Cassandra homepage.

If you encounter a bug or want to suggest an improvement, please create a ticket on the Spring Data issue tracker.

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<Person, Long> { ... }
   ```

2. Declare query methods on the interface.

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

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

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

@EnableJpaRepositories
class Config {}
```

or via XML configuration:

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

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

</beans>
```

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

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

4. Get the repository instance injected and use it.

```
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.

4.3.2. 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 6. 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 7. 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 8. Repository definitions using Domain Classes with Annotations

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

@Entity
public class Person {
  …
}

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

@Document
public 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 9. Repository definitions using Domain Classes with mixed Annotations

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

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

@Entity
@Document
public 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 10. Annotation-driven configuration of base packages

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

4.4. Defining query methods

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

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 11. 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);
```

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).

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 12. 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.

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 13. 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 14. 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 15. 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.

4.4.7. 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.

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 16. 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 17. 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. [1: JavaConfig in the Spring reference documentation]

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

Example 18. 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 19. 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 20. Interface for custom repository functionality

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

Example 21. 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 `JdbcTemplate`, take part in aspects, and so on.

Example 22. 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 23. 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 24. 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. To add custom behavior to all repositories, you first add an intermediate interface to declare the shared behavior.

Example 25. An interface declaring custom shared behavior

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

  void sharedCustomMethod(ID id);
}
```

Now your individual repository interfaces will extend this intermediate interface instead of the `Repository` interface to include the functionality declared. 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 26. 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(JpaEntityInformation entityInformation,
                          EntityManager entityManager) {
    super(entityInformation, entityManager);

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

  public void sharedCustomMethod(ID id) {
    // 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 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`.

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 27. 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 28. Configuring a custom repository base class using XML

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

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. 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 29. QueryDslPredicateExecutor interface

```
public interface QueryDslPredicateExecutor<T> {

    T findOne(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 30. 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);
```

4.7.2. 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 31. 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 32. 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 33. 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 34. 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, 0 indexed and defaults to 0.
`size`	Size of the page you want to retrieve, defaults to 20.
`sort`	Properties that should be sorted by in the format `property,property(,ASC|DESC)`. Default sort direction is ascending. Use multiple `sort` parameters if you want to switch directions, e.g. `?sort=firstname&sort=lastname,asc`.

To customize this behavior 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 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 35. 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.

Querydsl web support

For those stores having QueryDSL integration it is possible to derive queries from the attributes contained in a `Request` query string.

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

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

can be resolved to

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

using the `QuerydslPredicateArgumentResolver`.

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

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

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

```
@Controller
class UserController {

  @Autowired UserRepository repository;

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

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

    return "index";
  }
}
```

① Resolve query string arguments to matching `Predicate` for `User`.

The default binding is as follows:

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

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

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

  @Override
  default public 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.

4.7.3. 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 36. 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 37. 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 38. 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>
```

4.7.4. 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")
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";
  }
}
```

Reference Documentation

Document Structure

This part of the reference documentation explains the core functionality offered by Spring Data Cassandra.

Cassandra support introduces the Cassandra module feature set.

Cassandra repositories introduces the repository support for Cassandra.

Chapter 5. Cassandra support

The Cassandra support contains a wide range of features which are summarized below.

- Spring configuration support using Java based @Configuration classes or an XML namespace for a Cassandra driver instance and replica sets
- CassandraTemplate helper class that increases productivity performing common Cassandra operations. Includes integrated object mapping between CQL Tables and POJOs.
- Exception translation into Spring's portable Data Access Exception hierarchy
- Feature Rich Object Mapping integrated with Spring's Conversion Service
- Annotation based mapping metadata but extensible to support other metadata formats
- Persistence and mapping lifecycle events
- Java based Query, Criteria, and Update DSLs
- Automatic implementation of Repository interfaces including support for custom finder methods.

For most tasks you will find yourself using `CassandraTemplate` or the Repository support that both leverage the rich mapping functionality. CassandraTemplate is the place to look for accessing functionality such as incrementing counters or ad-hoc CRUD operations. CassandraTemplate also provides callback methods so that it is easy for you to get a hold of the low level API artifacts such as `com.datastax.driver.core.Session` to communicate directly with Cassandra. The goal with naming conventions on various API artifacts is to copy those in the base DataStax Java driver so you can easily map your existing knowledge onto the Spring APIs.

5.1. Getting Started

Spring Data Cassandra uses the DataStax Java Driver version 2.X, which supports DataStax Enterprise 4/Cassandra 2.0, and Java SE 6 or higher. The latest commercial release (2.X as of this writing) is recommended. An easy way to bootstrap setting up a working environment is to create a Spring based project in STS.

First you need to set up a running Cassandra server.

To create a Spring project in STS go to File → New → Spring Template Project → Simple Spring Utility Project → press Yes when prompted. Then enter a project and a package name such as org.spring.cassandra.example.

Then add the following to pom.xml dependencies section.

```
<dependencies>

  <!-- other dependency elements omitted -->

  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-cassandra</artifactId>
    <version>1.0.0.RELEASE</version>
  </dependency>

</dependencies>
```

Also change the version of Spring in the pom.xml to be

```
<spring.framework.version>3.2.8.RELEASE</spring.framework.version>
```

You will also need to add the location of the Spring Milestone repository for maven to your pom.xml which is at the same level of your <dependencies/> element

```
<repositories>
  <repository>
    <id>spring-milestone</id>
    <name>Spring Maven MILESTONE Repository</name>
    <url>http://repo.spring.io/libs-milestone</url>
  </repository>
</repositories>
```

The repository is also browseable here.

Create a simple Employee class to persist.

```
package org.spring.cassandra.example;

import org.springframework.data.cassandra.mapping.PrimaryKey;
import org.springframework.data.cassandra.mapping.Table;

@Table
public class Person {

 @PrimaryKey
 private String id;

 private String name;
 private int age;

 public Person(String id, String name, int age) {
  this.id = id;
  this.name = name;
  this.age = age;
 }

 public String getId() {
  return id;
 }

 public String getName() {
  return name;
 }

 public int getAge() {
  return age;
 }

 @Override
 public String toString() {
  return "Person [id=" + id + ", name=" + name + ", age=" + age + "]";
 }

}
```

And a main application to run

```
package org.spring.cassandra.example;

import java.net.InetAddress;
import java.net.UnknownHostException;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.data.cassandra.core.CassandraOperations;
import org.springframework.data.cassandra.core.CassandraTemplate;

import com.datastax.driver.core.Cluster;
import com.datastax.driver.core.Session;
import com.datastax.driver.core.querybuilder.QueryBuilder;
import com.datastax.driver.core.querybuilder.Select;

public class CassandraApp {

 private static final Logger LOG = LoggerFactory.getLogger(CassandraApp.class);

 private static Cluster cluster;
 private static Session session;

 public static void main(String[] args) {

  try {

   cluster = Cluster.builder().addContactPoints(InetAddress.getLocalHost()).build();

   session = cluster.connect("mykeyspace");

   CassandraOperations cassandraOps = new CassandraTemplate(session);

   cassandraOps.insert(new Person("1234567890", "David", 40));

   Select s = QueryBuilder.select().from("person");
   s.where(QueryBuilder.eq("id", "1234567890"));

   LOG.info(cassandraOps.queryForObject(s, Person.class).getId());

   cassandraOps.truncate("person");

  } catch (UnknownHostException e) {
   e.printStackTrace();
  }

 }
}
```

Even in this simple example, there are a few things to observe.

- You can create an instance of CassandraTemplate with a Cassandra Session, derived from the Cluster.
- You must annotate your POJO as a Cassandra @Table, and also annotate the @PrimaryKey. Optionally you can override these mapping names to match your Cassandra database table and column names.
- You can use CQL String, or the DataStax QueryBuilder to construct you queries.

5.2. Examples Repository

After the initial release of Spring Data Cassandra 1.0.0, we will start working on a showcase repository with full examples.

5.3. Connecting to Cassandra with Spring

5.3.1. Externalize Connection Properties

Create a properties file with the information you need to connect to Cassandra. The contact points are keyspace are the minimal required fields, but port is added here for clarity.

We will call this cassandra.properties

```
cassandra.contactpoints=10.1.55.80,10.1.55.81
cassandra.port=9042
cassandra.keyspace=showcase
```

We will use spring to load these properties into the Spring Context in the next two examples.

5.3.2. XML Configuration

The XML Configuration elements for a basic Cassandra configuration are shown below. These elements all use default bean names to keep the configuration code clean and readable.

While this example show how easy it is to configure Spring to connect to Cassandra, there are many other options. Basically, any option available with the DataStax Java Driver is also available in the Spring Data Cassandra configuration. This is including, but not limited to Authentication, Load Balancing Policies, Retry Policies and Pooling Options. All of the Spring Data Cassandra method names and XML elements are named exactly (or as close as possible) like the configuration options on the driver so mapping any existing driver configuration should be straight forward.

```
<?xml version='1.0'?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:cassandra=
"http://www.springframework.org/schema/data/cassandra"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/cql
http://www.springframework.org/schema/cql/spring-cql-1.0.xsd
    http://www.springframework.org/schema/data/cassandra
http://www.springframework.org/schema/data/cassandra/spring-cassandra-1.0.xsd
    http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.2.xsd">

  <!-- Loads the properties into the Spring Context and uses them to fill
    in placeholders in the bean definitions -->
  <context:property-placeholder location="classpath:cassandra.properties" />

  <!-- REQUIRED: The Cassandra Cluster -->
  <cassandra:cluster contact-points="${cassandra.contactpoints}"
    port="${cassandra.port}" />

  <!-- REQUIRED: The Cassandra Session, built from the Cluster, and attaching
    to a keyspace -->
  <cassandra:session keyspace-name="${cassandra.keyspace}" />

  <!-- REQUIRED: The Default Cassandra Mapping Context used by CassandraConverter -->
  <cassandra:mapping />

  <!-- REQUIRED: The Default Cassandra Converter used by CassandraTemplate -->
  <cassandra:converter />

  <!-- REQUIRED: The Cassandra Template is the building block of all Spring
    Data Cassandra -->
  <cassandra:template id="cassandraTemplate" />

  <!-- OPTIONAL: If you are using Spring Data Cassandra Repositories, add
    your base packages to scan here -->
  <cassandra:repositories base-package="org.spring.cassandra.example.repo" />

</beans>
```

5.3.3. Java Configuration

The following class show a basic and minimal Cassandra configuration using the AnnotationConfigApplicationContext (aka JavaConfig).

```
package org.spring.cassandra.example.config;
```

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.core.env.Environment;
import org.springframework.data.cassandra.config.CassandraClusterFactoryBean;
import org.springframework.data.cassandra.config.CassandraSessionFactoryBean;
import org.springframework.data.cassandra.config.SchemaAction;
import org.springframework.data.cassandra.convert.CassandraConverter;
import org.springframework.data.cassandra.convert.MappingCassandraConverter;
import org.springframework.data.cassandra.core.CassandraOperations;
import org.springframework.data.cassandra.core.CassandraTemplate;
import org.springframework.data.cassandra.mapping.BasicCassandraMappingContext;
import org.springframework.data.cassandra.mapping.CassandraMappingContext;
import
org.springframework.data.cassandra.repository.config.EnableCassandraRepositories;

@Configuration
@PropertySource(value = { "classpath:cassandra.properties" })
@EnableCassandraRepositories(basePackages = { "org.spring.cassandra.example.repo" })
public class CassandraConfig {

  private static final Logger LOG = LoggerFactory.getLogger(CassandraConfig.class);

  @Autowired
  private Environment env;

  @Bean
  public CassandraClusterFactoryBean cluster() {

    CassandraClusterFactoryBean cluster = new CassandraClusterFactoryBean();
    cluster.setContactPoints(env.getProperty("cassandra.contactpoints"));
    cluster.setPort(Integer.parseInt(env.getProperty("cassandra.port")));

    return cluster;
  }

  @Bean
  public CassandraMappingContext mappingContext() {
    return new BasicCassandraMappingContext();
  }

  @Bean
  public CassandraConverter converter() {
    return new MappingCassandraConverter(mappingContext());
  }

  @Bean
  public CassandraSessionFactoryBean session() throws Exception {
```

```
    CassandraSessionFactoryBean session = new CassandraSessionFactoryBean();
    session.setCluster(cluster().getObject());
    session.setKeyspaceName(env.getProperty("cassandra.keyspace"));
    session.setConverter(converter());
    session.setSchemaAction(SchemaAction.NONE);

    return session;
  }

  @Bean
  public CassandraOperations cassandraTemplate() throws Exception {
    return new CassandraTemplate(session().getObject());
  }
}
```

5.4. General auditing configuration

Auditing support is not available in the current version.

5.5. Introduction to CassandraTemplate

5.5.1. Instantiating CassandraTemplate

`CassandraTemplate` should always be configured as a Spring Bean, although we show an example above where you can instantiate it directly. But for the purposes of this being a Spring module, lets assume we are using the Spring Container.

`CassandraTemplate` is an implementation of `CassandraOperations`. You should always assign your `CassandraTemplate` to its interface definition, `CassandraOperations`.

There are 2 easy ways to get a `CassandraTemplate`, depending on how you load you Spring Application Context.

AutoWiring

```
@Autowired
private CassandraOperations cassandraOperations;
```

Like all Spring Autowiring, this assumes there is only one bean of type `CassandraOperations` in the `ApplicationContext`. If you have multiple `CassandraTemplate` beans (which will be the case if you are working with multiple keyspaces in the same project), use the `@Qualifier`annotation to designate which bean you want to Autowire.

```
@Autowired
@Qualifier("myTemplateBeanId")
private CassandraOperations cassandraOperations;
```

Bean Lookup with ApplicationContext

You can also just lookup the `CassandraTemplate` bean from the `ApplicationContext`.

```
CassandraOperations cassandraOperations = applicationContext.getBean(
"cassandraTemplate", CassandraOperations.class);
```

5.6. Saving, Updating, and Removing Rows

`CassandraTemplate` provides a simple way for you to save, update, and delete your domain objects and map those objects to documents stored in Cassandra.

5.6.1. How the Composite Primary Key fields are handled in the mapping layer

Cassandra requires that you have at least 1 Partition Key field for a CQL Table. Alternately, you can have one or more Clustering Key fields. When your CQL Table has a composite Primary Key field you must create a @PrimaryKeyClass to define the structure of the composite PK. In this context, composite PK means one or more partition columns, or 1 partition column plus one or more clustering columns.

Simplest Composite Key

The simplest for of a Composite key is a key with one partition key and one clustering key. Here is an example of a CQL Table, and the corresponding POJOs that represent the table and it's composite key.

CQL Table defined in Cassandra

```
create table login_event(
  person_id text,
  event_time timestamp,
  event_code int,
  ip_address text,
  primary key (person_id, event_time))
  with CLUSTERING ORDER BY (event_time DESC)
;
```

Class defining the **Composite Primary Key**.

NOTE	PrimaryKeyClass must implement `Serializable` and provide implementation of `hashCode()` and `equals()` just like the example.

```
package org.spring.cassandra.example;

import java.io.Serializable;
import java.util.Date;

import org.springframework.cassandra.core.Ordering;
import org.springframework.cassandra.core.PrimaryKeyType;
import org.springframework.data.cassandra.mapping.PrimaryKeyClass;
import org.springframework.data.cassandra.mapping.PrimaryKeyColumn;

@PrimaryKeyClass
public class LoginEventKey implements Serializable {

  @PrimaryKeyColumn(name = "person_id", ordinal = 0, type = PrimaryKeyType.
PARTITIONED)
  private String personId;

  @PrimaryKeyColumn(name = "event_time", ordinal = 1, type = PrimaryKeyType.CLUSTERED,
ordering = Ordering.DESCENDING)
  private Date eventTime;

  public String getPersonId() {
    return personId;
  }

  public void setPersonId(String personId) {
    this.personId = personId;
  }

  public Date getEventTime() {
    return eventTime;
  }

  public void setEventTime(Date eventTime) {
    this.eventTime = eventTime;
  }

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((eventTime == null) ? 0 : eventTime.hashCode());
    result = prime * result + ((personId == null) ? 0 : personId.hashCode());
    return result;
  }

  @Override
```

```
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    LoginEventKey other = (LoginEventKey) obj;
    if (eventTime == null) {
      if (other.eventTime != null)
        return false;
    } else if (!eventTime.equals(other.eventTime))
      return false;
    if (personId == null) {
      if (other.personId != null)
        return false;
    } else if (!personId.equals(other.personId))
      return false;
    return true;
  }
}
```

Class defining the CQL Table, having the **Composite Primary Key** as an attribute and annotated as the `PrimaryKey`.

```
package org.spring.cassandra.example;

import org.springframework.data.cassandra.mapping.Column;
import org.springframework.data.cassandra.mapping.PrimaryKey;
import org.springframework.data.cassandra.mapping.Table;

@Table(value = "login_event")
public class LoginEvent {

  @PrimaryKey
  private LoginEventKey pk;

  @Column(value = "event_code")
  private int eventCode;

  @Column(value = "ip_address")
  private String ipAddress;

  public LoginEventKey getPk() {
    return pk;
  }

  public void setPk(LoginEventKey pk) {
    this.pk = pk;
  }

  public int getEventCode() {
    return eventCode;
  }

  public void setEventCode(int eventCode) {
    this.eventCode = eventCode;
  }

  public String getIpAddress() {
    return ipAddress;
  }

  public void setIpAddress(String ipAddress) {
    this.ipAddress = ipAddress;
  }

}
```

Complex Composite Primary Key

The annotations provided with Spring Data Cassandra can handle any key combination available in Cassandra. Here is one more example of a Composite Primary Key with 5 columns, 2 of which are a composite partition key, and the remaining 3 are ordered clustering keys. The getters/setters,

hashCode and equals are omitted for brevity.

```
package org.spring.cassandra.example;

import java.io.Serializable;
import java.util.Date;

import org.springframework.cassandra.core.Ordering;
import org.springframework.cassandra.core.PrimaryKeyType;
import org.springframework.data.cassandra.mapping.PrimaryKeyClass;
import org.springframework.data.cassandra.mapping.PrimaryKeyColumn;

@PrimaryKeyClass
public class DetailedLoginEventKey implements Serializable {

  @PrimaryKeyColumn(name = "person_id", ordinal = 0, type = PrimaryKeyType.
PARTITIONED)
  private String personId;

  @PrimaryKeyColumn(name = "wks_id", ordinal = 1, type = PrimaryKeyType.PARTITIONED)
  private String workstationId;

  @PrimaryKeyColumn(ordinal = 2, type = PrimaryKeyType.CLUSTERED, ordering = Ordering
.ASCENDING)
  private Date application;

  @PrimaryKeyColumn(name = "event_code", ordinal = 3, type = PrimaryKeyType.CLUSTERED,
ordering = Ordering.ASCENDING)
  private Date eventCode;

  @PrimaryKeyColumn(name = "event_time", ordinal = 4, type = PrimaryKeyType.CLUSTERED,
ordering = Ordering.DESCENDING)
  private Date eventTime;

  ...

}
```

5.6.2. Type mapping

Spring Data Cassandra relies on the DataStax Java Driver type mapping component. This approach ensures that as types are added or changed, the Spring Data Cassandra module will continue to function without requiring changes. For more information on the DataStax CQL3 to Java Type mappings, please see their Documentation here.

5.6.3. Methods for saving and inserting rows

Single records inserts

To insert one row at a time, there are many options. At this point you should already have a cassandraTemplate available to you so we will just how the relevant code for each section, omitting the template setup.

Insert a record with an annotated POJO.

```
cassandraOperations.insert(new Person("123123123", "Alison", 39));
```

Insert a row using the QueryBuilder.Insert object that is part of the DataStax Java Driver.

```
Insert insert = QueryBuilder.insertInto("person");
insert.setConsistencyLevel(ConsistencyLevel.ONE);
insert.value("id", "123123123");
insert.value("name", "Alison");
insert.value("age", 39);

cassandraOperations.execute(insert);
```

Then there is always the old fashioned way. You can write your own CQL statements.

```
String cql = "insert into person (id, name, age) values ('123123123', 'Alison', 39)";

cassandraOperations.execute(cql);
```

Multiple inserts for high speed ingestion

CQLOperations, which is extended by CassandraOperations is a lower level Template that you can use for just about anything you need to accomplish with Cassandra. CqlOperations includes several overloaded methods named `ingest()`.

Use these methods to pass a CQL String with Bind Markers, and your preferred flavor of data set (Object[][] and List<List<T>>).

The ingest method takes advantage of static PreparedStatements that are only prepared once for performance. Each record in your data list is bound to the same PreparedStatement, then executed asynchronously for high performance.

```
String cqlIngest = "insert into person (id, name, age) values (?, ?, ?)";

List<Object> person1 = new ArrayList<Object>();
person1.add("10000");
person1.add("David");
person1.add(40);

List<Object> person2 = new ArrayList<Object>();
person2.add("10001");
person2.add("Roger");
person2.add(65);

List<List<?>> people = new ArrayList<List<?>>();
people.add(person1);
people.add(person2);

cassandraOperations.ingest(cqlIngest, people);
```

5.6.4. Updating rows in a CQL table

Much like inserting, there are several flavors of update from which you can choose.

Update a record with an annotated POJO.

```
cassandraOperations.update(new Person("123123123", "Alison", 35));
```

Update a row using the QueryBuilder.Update object that is part of the DataStax Java Driver.

```
Update update = QueryBuilder.update("person");
update.setConsistencyLevel(ConsistencyLevel.ONE);
update.with(QueryBuilder.set("age", 35));
update.where(QueryBuilder.eq("id", "123123123"));

cassandraOperations.execute(update);
```

Then there is always the old fashioned way. You can write your own CQL statements.

```
String cql = "update person set age = 35 where id = '123123123'";

cassandraOperations.execute(cql);
```

5.6.5. Methods for removing rows

Much like inserting, there are several flavors of delete from which you can choose.

Delete a record with an annotated POJO.

```
cassandraOperations.delete(new Person("123123123", null, 0));
```

Delete a row using the QueryBuilder.Delete object that is part of the DataStax Java Driver.

```
Delete delete = QueryBuilder.delete().from("person");
delete.where(QueryBuilder.eq("id", "123123123"));

cassandraOperations.execute(delete);
```

Then there is always the old fashioned way. You can write your own CQL statements.

```
String cql = "delete from person where id = '123123123'";

cassandraOperations.execute(cql);
```

5.6.6. Methods for truncating tables

Much like inserting, there are several flavors of truncate from which you can choose.

Truncate a table using the truncate() method.

```
cassandraOperations.truncate("person");
```

Truncate a table using the QueryBuilder.Truncate object that is part of the DataStax Java Driver.

```
Truncate truncate = QueryBuilder.truncate("person");

cassandraOperations.execute(truncate);
```

Then there is always the old fashioned way. You can write your own CQL statements.

```
String cql = "truncate person";

cassandraOperations.execute(cql);
```

5.7. Querying CQL Tables

Tthere are several flavors of select and query from which you can choose. Please see the CassandraTemplate API documentation for all overloads available.

Query a table for multiple rows and map the results to a POJO.

```
String cqlAll = "select * from person";

List<Person> results = cassandraOperations.select(cqlAll, Person.class);
for (Person p : results) {
    LOG.info(String.format("Found People with Name [%s] for id [%s]", p.getName(), p
.getId()));
}
```

Query a table for a single row and map the result to a POJO.

```
String cqlOne = "select * from person where id = '123123123'";

Person p = cassandraOperations.selectOne(cqlOne, Person.class);
LOG.info(String.format("Found Person with Name [%s] for id [%s]", p.getName(), p.
getId()));
```

Query a table using the QueryBuilder.Select object that is part of the DataStax Java Driver.

```
Select select = QueryBuilder.select().from("person");
select.where(QueryBuilder.eq("id", "123123123"));

Person p = cassandraOperations.selectOne(select, Person.class);
LOG.info(String.format("Found Person with Name [%s] for id [%s]", p.getName(), p.
getId()));
```

Then there is always the old fashioned way. You can write your own CQL statements, and there are several callback handlers for mapping the results. The example uses the RowMapper interface.

```
String cqlAll = "select * from person";
List<Person> results = cassandraOperations.query(cqlAll, new RowMapper<Person>() {

    public Person mapRow(Row row, int rowNum) throws DriverException {
        Person p = new Person(row.getString("id"), row.getString("name"), row.getInt(
"age"));
        return p;
    }
});

for (Person p : results) {
    LOG.info(String.format("Found People with Name [%s] for id [%s]", p.getName(), p
.getId()));
}
```

5.8. Overriding default mapping with custom converters

In order to have more fine grained control over the mapping process you can register Spring converters with the `CassandraConverter` implementations such as the `MappingCassandraConverter`.

The `MappingCassandraConverter` checks to see if there are any Spring converters that can handle a specific class before attempting to map the object itself. To 'hijack' the normal mapping strategies of the `MappingCassandraConverter`, perhaps for increased performance or other custom mapping needs, you first need to create an implementation of the Spring `Converter` interface and then register it with the MappingConverter.

NOTE	For more information on the Spring type conversion service see the reference docs here.

5.8.1. Saving using a registered Spring Converter

Coming Soon!

5.8.2. Reading using a Spring Converter

Coming Soon!

5.8.3. Registering Spring Converters with the CassandraConverter

Coming Soon!

5.8.4. Converter disambiguation

Coming Soon!

5.9. Executing Commands

5.9.1. Methods for executing commands

The CassandraTemplate has many overloads for execute() and executeAsync(). Pass in the CQL command you wish to be executed, and handle the appropriate response.

This example uses the basic AsynchronousQueryListener that comes with Spring Data Cassandra. Please see the API documentation for all the options. There should be nothing you cannot perform in Cassandra with the execute() and executeAsync() methods.

```
cassandraOperations.executeAsynchronously("delete from person where id = '123123123'",
        new AsynchronousQueryListener() {

            public void onQueryComplete(ResultSetFuture rsf) {
                LOG.info("Async Query Completed");
            }
        });
```

This example shows how to create and drop a table, using different API objects, all passed to the execute() methods.

```
cassandraOperations.execute("create table test_table (id uuid primary key, event
text)");

DropTableSpecification dropper = DropTableSpecification.dropTable("test_table");
cassandraOperations.execute(dropper);
```

5.10. Exception Translation

The Spring framework provides exception translation for a wide variety of database and mapping technologies. This has traditionally been for JDBC and JPA. The Spring support for Cassandra extends this feature to the Cassandra Database by providing an implementation of the `org.springframework.dao.support.PersistenceExceptionTranslator` interface.

The motivation behind mapping to Spring's consistent data access exception hierarchy is that you are then able to write portable and descriptive exception handling code without resorting to coding against Cassandra Exceptions. All of Spring's data access exceptions are inherited from the root `DataAccessException` class so you can be sure that you will be able to catch all database related exception within a single try-catch block.

Chapter 6. Cassandra repositories

6.1. Introduction

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

6.2. Usage

To access domain entities stored in a Cassandra you can leverage our sophisticated repository support that eases implementing those quite significantly. To do so, simply create an interface for your repository:

TODO

6.3. Query methods

6.3.1. Repository delete queries

6.4. Miscellaneous

6.4.1. CDI Integration

The Spring Data Cassandra CDI extension will pick up the `CassandraTemplate` available as CDI bean and create a proxy for a Spring Data repository whenever an bean of a repository type is requested by the container. Thus obtaining an instance of a Spring Data repository is a matter of declaring an `@Inject`-ed property:

```
class RepositoryClient {

  @Inject
  PersonRepository repository;

  public void businessMethod() {

    List<Person> people = repository.findAll();
  }
}
```

Chapter 7. Mapping

Rich mapping support is provided by the `CassandraMappingConverter` . `CassandraMappingConverter` has a rich metadata model that provides a full feature set of functionality to map domain objects to CQL Tables. The mapping metadata model is populated using annotations on your domain objects. However, the infrastructure is not limited to using annotations as the only source of metadata information. The `CassandraMappingConverter` also allows you to map objects to documents without providing any additional metadata, by following a set of conventions.

In this section we will describe the features of the CassandraMappingConverter. How to use conventions for mapping objects to documents and how to override those conventions with annotation based mapping metadata.

7.1. Convention based Mapping

`CassandraMappingConverter` has a few conventions for mapping objects to CQL Tables when no additional mapping metadata is provided. The conventions are:

- The short Java class name is mapped to the table name in the following manner. The class `com.bigbank.SavingsAccount` maps to `savings_account` table name.
- The converter will use any Spring Converters registered with it to override the default mapping of object properties to document field/values.
- The fields of an object are used to convert to and from fields in the document. Public JavaBean properties are not used.

7.1.1. How the CQL Composite Primary Key fields are handled in the mapping layer

TODO

7.1.2. Mapping Configuration

Unless explicitly configured, an instance of `CassandraMappingConverter` is created by default when creating a `CassandraTemplate` . You can create your own instance of the `MappingCassandraConverter` so as to tell it where to scan the classpath at startup your domain classes in order to extract metadata and construct indexes. Also, by creating your own instance you can register Spring converters to use for mapping specific classes to and from the database.

You can configure the `CassandraMappingConverter` and CassandraTemplate either using Java or XML based metadata. Here is an example using Spring's Java based configuration

Example 39. @Configuration class to configure Cassandra mapping support

```
TODO
```

Example 40. XML schema to configure Cassandra mapping support

```
TODO
```

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 2. Attributes

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

Appendix B: Populators namespace reference

The <populator /> element

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

Table 3. 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 4. 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`
`REGEX`	`Regex`, `MatchesRegex`, `Matches`
`STARTING_WITH`	`StartingWith`, `IsStartingWith`, `StartsWith`
`TRUE`	`True`, `IsTrue`

Logical keyword	Keyword expressions
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 5. 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`.
`Future<T>`	A `Future`. Expects method to be annotated with `@Async` and requires Spring's asynchronous method execution capability enabled.
`CompletableFuture<T>`	A Java 8 `CompletableFuture`. Expects method to be annotated with `@Async` and requires Spring's asynchronous method execution capability enabled.
`ListenableFuture`	A `org.springframework.util.concurrent.ListenableFuture`. Expects method to be annotated with `@Async` and requires Spring's asynchronous method execution capability enabled.
`Slice`	A sized chunk of data with information whether there is more data available. Requires a `Pageable` method parameter.
`Page<T>`	A `Slice` with additional information, e.g. the total number of results. Requires a `Pageable` method parameter.

Return type	Description
`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.